# Programa Educativo Para Colegios Secundarios

## Area Computación

## Tomo 2






## PROGRAMA EDUCATIVO
## PARA COLEGIOS SECUNDARIOS

# AREA COMPUTACION

### TOMO 2

## EQUIPO DE PRODUCCION

**VIVIANA RUBINSTEIN**
Computadora Científica (UBA)

**REGINA R. DZEGHITMAN DE RUBINSTEIN**
Profesora Licenciada en Ciencias de la Educación (UBA)

**SUSANA SIERRA**
Analista de Sistemas (UB)

## ASESORES

**HORACIO RODRIGUEZ**
Contador Público Nacional (UBA)

**PAULINA C. SOCOLOVSKY DE FRENKEL**
Computadora Científica (UBA)

Realización Gráfica: **ALBERTO BENETTI**

Edita:

Aswork s.a.

# INDICE

## UNIDAD 7

Esperamos que a través del camino que hemos recorrido juntos lograron Uds. comprender los objetivos generales de las Unidades temáticas estudiadas. Que la sucesión de sus experiencias de pensamiento y acción significaron para Uds. un proceso de aprendizaje cuyo producto es un modelo útil y perfectible que sirve para representar y resolver problemas.

Ubicamos a la Computación en la Historia de la Ciencia y de la Técnica, a la Máquina como un Instrumento más para resolver problemas.

Presentamos las etapas del proceso de Computación como un modelo del pensamiento para acceder al conocimiento necesario y responder adecuadamente a las necesidades individuales y sociales.

Les proporcionamos una actividad para que Uds. logren integrar el Pensar y el Hacer a través del aprendizaje de TIMBA.

Ampliaron luego su marco teórico cuando desarrollaron el estudio correspondiente a Programación Estructurada y Estructura de Datos.

Están Uds. ahora en condiciones de aprender un nuevo lenguaje de programación, comprendiendo que programar es mucho más que utilizar un determinado lenguaje. Atravesamos una etapa histórica donde asiduamente nacen lenguajes y debemos estar preparados para acceder en algún momento a ellos.

Siempre que esten frente a uno nuevo deberán preguntarse: ¿qué estructuras de datos maneja?, ¿qué sentencias operativas?, ¿qué sentencias de control? ...

Deben Uds. concentrar esfuerzos en el diseño de algoritmos, utilizando la metodologia planteada, para que ya desarrollado traduzcan Uds. el algoritmo al lenguaje disponible.

Brindamos ahora a Uds. la posibilidad de acceder al conocimiento y aplicación de un lenguaje de programación rico y complejo: BASIC, y al mismo tiempo una mayor comprensión del pensar y el quehacer de las personas dedicadas a la Computación.

Con la lectura y análisis de este texto, y posteriores discusiones en grupo, conducidos por su coordinador, esperamos que Uds. logren cumplir los siguientes.

## • OBJETIVOS GENERALES

1. Conozcan las características generales del lenguaje Basic.
2. Conozcan la estructura de un programa escrito en lenguaje Basic.
3. Comprendan la interrelación entre los aspectos estáticos y dinámicos de un programa Basic.
4. Conozcan las funciones de las sentencias Basic.
5. Comprendan el significado de los tipos de Condición en Basic.
6. Comprendan la función específica de cada Comando.
7. Conozcan la estructura de datos denominada Arreglo.
8. Comprendan la riqueza de posibilidades de esta estructura de datos.
9. Comprendan la ventaja del uso de subrutinas.
10. Valoren en su justa medida la importancia de esta profesión.

## • OBJETIVOS ESPECIFICOS

1.1. Describan los elementos del lenguaje Basic.

2.1. Dado un Problema, planteen un algoritmo y lo codifiquen.

3.1. Analicen la información necesaria para comenzar a programar.

3.2. Expliquen la diferencia entre sentencia y comando.

4.1. Describan las sentencias Basic.

5.1. Expliquen la definición de Condición.

6.1. Analicen la función de cada comando.

6.2. Expliciten el formato general de cada comando.

7.1. Caractericen un Arreglo.

7.2. Clasifiquen esta estructura de datos usando como criterio la dimensión.

7.3. Describan las convenciones para nombrar un arreglo.

8.1. Propongan un problema para ilustrar las ventajas de la aplicación de esta estructura de datos.

9.1. Describan las características de la sentencia GOSUB.

9.2. Apliquen la sentencia GOSUB en programas donde resulte ventajoso.

10.1. Participen activamente en el analisis de las afirmaciones acerca del trabajo del programador en el texto correspondiente a Prueba y Corrección de programas.

## 1. INTRODUCCION

En esta unidad comenzaremos a estudiar un lenguaje muy conocido y difundido que se llama BASIC (Beginner's All purpose Symbolic Instruction Code = código de instrucción simbólica de utilidad general para principiantes).

Fue desarrollado originariamente a mediados de la década del 60 por John Kemeny y Thomas Kurtz en Dartmouth Collage.

Es un lenguaje simple que utiliza palabras en Inglés (PRINT, LET, WHILE, LIST, etc.) y algunas instrucciones nos recuerdan las fórmulas del álgebra.

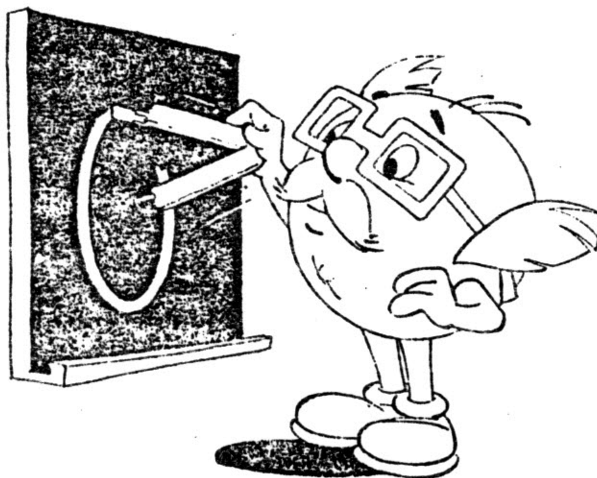
Hay varios niveles de BASIC, nosotros comenzaremos estudiando un subconjunto elemental de instrucciones.

Un programa en BASIC consta de instrucciones que se escriben como sentencias separadas. Cada sentencia comienza en un renglón diferente y tiene un número que la identifica (NUMERO DE LINEA). A continuación del número va una PALABRA CLAVE que indica el tipo de operación a ejecutar.

Se pueden incluir espacios en blanco, separando las palabras, para hacer más legible la sentencia.

La numeración de las líneas debe ser creciente.

Supongamos que queremos obtener el área de un círculo. La fórmula para calcularla es: $\pi \times R^2$, donde R es el radio del círculo.



¿Qué instrucciones debemos escribir?¿Cuál será nuestro algoritmo?.

Primero, escribimos una instrucción que permita darle al computador el valor del radio; luego, otra instrucción para poder plantear la fórmula, calcular el área y guardarla en la memoria; finalmente una instrucción que permita visualizar el resultado.

En BASIC, esto puede lograrse de la siguiente manera:

```
10 INPUT RADIO

20 LET AREA = 3.1416*RADIO^2

30 PRINT RADIO, AREA
```

Analicemos este programa:

Cada sentencia está numerada, tiene una PALABRA CLAVE y la numeración es ascendente.

La primer sentencia permite ingresar a través de INPUT (ingrese) un valor numérico del RADIO desde el teclado. La segunda hace que se le asigne a la variable AREA el resultado del cálculo de 3.1416 x $RADIO^2$. Por último, la tercera muestra en la pantalla los valores del RADIO y del AREA.

Sólo nos falta una instrucción que indique el fin del programa:

```
40 END
```

En la sentencia 20 aparecen dos símbolos que indican la multiplicación (*) y la potenciación (^). Sin entrar a analizar en detalle todavía estas instrucciones, intentemos descubrir, con lo poco que vimos hasta ahora, que hacen los siguientes programas:

A.

```
10 INPUT A,B,C,D,
20 LET X = (A + B + C + D)/4
30 PRINT X
40 END
```

B.

```
100 INPUT A,B
200 LET X = A + B
300 LET Y = A – B
400 LET T = A*B
500 LET Z = A/B
600 PRINT X,Y,T,Z
700 END
```



¿Nos animamos a hacer un programa en BASIC que calcule el área de un rectángulo conociendo el valor de su BASE y su ALTURA?

## 2. LOS ELEMENTOS DE BASIC

En TIMBA, vimos que las instrucciones actúan sobre los naipes o las cartas, agrupados en estructuras de pilas. Estas instrucciones ordenan TOMAR, INVERTIR o DEPOSITAR cartas.

Estas son los elementos naturales sobre los que actúan las instrucciones, alterando de un modo u otro el estado de las pilas o de la carta.

En BASIC, las instrucciones actúan sobre un conjunto mucho más rico y complejo de elementos: cantidades aritméticas, palabras, letras, símbolos, etc. Conocer estos elementos significa iniciarse en el conocimiento de este lenguaje.

Los veremos a cada uno por separado:

### A. CONSTANTES

Las constantes son valores que el BASIC usa durante la ejecución de programas.

Hay dos tipos de constantes: NUMERICAS y ALFANUMERICAS.

* NUMERICAS

Las constantes numéricas pueden expresarse como enteros o decimales.

Una constante numérica puede estar precedida por un signo + ó –, asumiendo que es positiva cuando no tiene signo.

Los siguientes son ejemplos de constantes numéricas:

```
  3
+3
-3
  128
-1.456
  0
```

* ALFANUMERICAS O CADENAS DE CARACTERES

Así como podemos representar valores numéricos, es posible también escribir palabras o sucesiones de caracteres (letras, números o caracteres especiales), que se conocen con el nombre de STRINGS (cadenas).

De esta forma escribimos nombres, direcciones o cualquier mensaje textual. Por ejemplo:

```
"APRENDO A PROGRAMAR"
"$13569"
"LA RESPUESTA ES:"
"JUAN PECOS ES UN BUEN HOMBRE"
```



Estas cadenas son secuencias de hasta 255 caracteres alfanuméricos encerrados entre comillas.

**B. VARIABLES**

El concepto de VARIABLE es muy importante en programación. Es similar, pero no idéntico a la idea de variable que usamos en álgebra.

Una variable es un símbolo que representa un valor, y las expresiones pueden escribirse usando variables en lugar de los valores que ellas representan. Por ejemplo, el valor de la expresión:

X + Y

depende de los valores que X e Y representen en determinado momento.

En programación, una variable tiene también una interpretación física: es un lugar en la memoria de la computadora que almacena un VALOR y tiene un NOMBRE o identificador que sirve para referirnos a ese LUGAR y a ese VALOR. Por ejemplo:

X = 20

representa un lugar en la memoria: X es el nombre asociado con ese lugar y 20 es el valor que ese lugar almacena en este momento.

Decimos entonces que una VARIABLE nombra un lugar de la memoria, que tiene un NOMBRE y almacena un VALOR y ese valor puede variar durante la ejecución del programa.

El valor de la variable puede ser asignado explícitamente por el programador, o como resultado de un cálculo del programa.

En definitiva una variable contiene una constante en un momento dado de la ejecución.

Antes que se le asigne un valor a una variable, el BASIC asume que es cero.

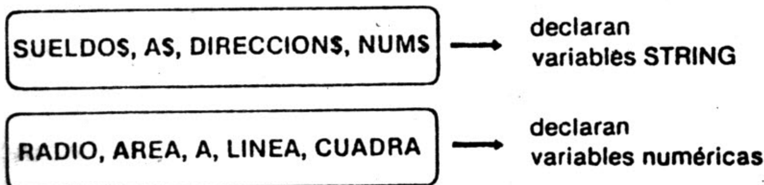
Existen reglas o convenciones para escribir el nombre de una variable:

- No puede ser una PALABRA CLAVE
- El primer caracter debe ser una letra
- Puede tener cualquier cantidad de caracteres (es decir, letras y números).

Como a las variables les podemos asignar constantes numéricas o alfanuméricas, existe la convención que los nombres de las variables que representan constantes alfanuméricas (STRING) se escriben con el signo pesos ($) como último caracter.

Si el nombre no termina de esta manera, se asume que es una variable que representa una constante numérica.

Por ejemplo:




## C. OPERACIONES ARITMETICAS

En BASIC se utilizan símbolos especiales llamados OPERADORES para plantear las operaciones aritméticas. Estos son:

+ para la suma

– para la resta

* para la multiplicación

/ para la división

^ para la exponenciación o potenciación

Estos operadores se usan para relacionar constantes y/o variables y de esta forma generar fórmulas o expresiones. Las operaciones se efectuan con los valores numéricos de la expresión, dando como resultado un único valor numérico.

Por ejemplo:

$A + B/2 - 3.84 \times C^2$

$B^2 - 4 \times A \times C$

$(A - B)/C \times 2.183 - D^2/4 + 1 \times F$

se escribe

```
A + B/2 - 3.84*C^2
B^2 - 4*A*C
(A - B)/C*2.183 - (D^2)/4 + 1*F
```

Cuando en una expresión hay varios operadores es necesario conocer el orden de ejecución o la jerarquía de las operaciones antes de obtener el resultado.

Por ejemplo:

A/B*C puede interpretarse como A/(BxC) o (A/B)xC



En programación, la jerarquía es la siguiente:

* primero se ejecutan las operaciones de POTENCIACION
* luego, las de MULTIPLICACION y DIVISION
* por último, las de SUMA y RESTA

Pero, dentro de un mismo grupo jerárquico las operaciones se ejecutan de IZQUIERDA a DERECHA.

Así:

A/B*C se interpreta como la expresión matemática: (A/B)xC

```
M^2 + B/A – C*4.18
```

equivale a la expresión matemática: $M^2 + B/A - 4.18 \times C$
y se efectúa:

primero, M^2

a continuación, B/A

luego, 4.18*C

luego, la suma

y por último, la resta

Para modificar este orden jerárquico normal de las operaciones se colocan pares de paréntesis, de forma tal que las operaciones incluídas entre los paréntesis más internos se efectúan primero, luego los segundos más internos y así sucesivamente.

Por ejemplo, si la expresión que debemos calcular es:

$[(B + A)^2 + (3xE)] ((M + N)/2)$

Entonces en BASIC, escribimos:

```
[(B + A)^2 + (3*E)]^((M + N)/2
```

Resumiendo: las constantes, las variables y los operadores son los elementos sobre los que actúan las sentencias o instrucciones de BASIC.

Los programas son almacenados en la memoria para luego poder ejecutarlos. ¿Qué pasa si vamos escribiendo línea tras línea? Cada una se va almacenando en la memoria. Luego, cuando terminamos con todas las líneas y estamos listos para ejecutar el programa, escribimos el comando **RUN** y presionamos la tecla ENTER o RETURN.

Así comienza la ejecución de las sentencias a partir del menor número de línea.

La computadora responde a:

a - sentencias

b - comandos

La diferencia entre sentencias y comandos es que las sentencias se ejecutan desde un programa y los comandos son ejecutados solamente desde el teclado. Por ejemplo: si escribimos NEW y presionamos la tecla ENTER o RETURN, el comando NEW borra la memoria de la computadora.

## NUMERO DE LINEA

Las sentencias de un programa se numeran y se almacenan en la memoria de la computadora.

El número que le damos a la línea se llama NUMERO DE LINEA y tiene dos funciones. La primera, es la de identificar a la línea como parte de un programa. Todas las líneas de un programa deben tener un número de línea. La segunda es la de indicar la posición de la línea dentro de un programa. La línea de menor número es la primera del programa y la línea de mayor número es la última del programa. Cuando se ejecuta un programa la línea de menor número es la primera que se ejecuta.

Para almacenar las líneas de un programa, ingresamos el número de línea, la sentencia y luego presionamos la tecla ENTER o RETURN. Probemos esto:

```
10 PRINT "HOLA"
20 END
```

Ya tenemos almacenadas dos líneas de un programa. Ahora ejecutémoslo.

El orden en que ingresamos las líneas no tiene importancia, ya que se guardan en la secuencia correcta cuando presionamos ENTER o RETURN. Por ejemplo, en el programa anterior, si queremos agregar otra sentencia **PRINT**, para que aparezca algo después de HOLA, podemos elegir un número entre 10 y 20 (podría ser 15). Hagámoslo.

```
15 PRINT "CHAU"
```

Ahora tenemos tres líneas de programa. La computadora automáticamente insertó la última línea ingresada entre las líneas 10 y 20.

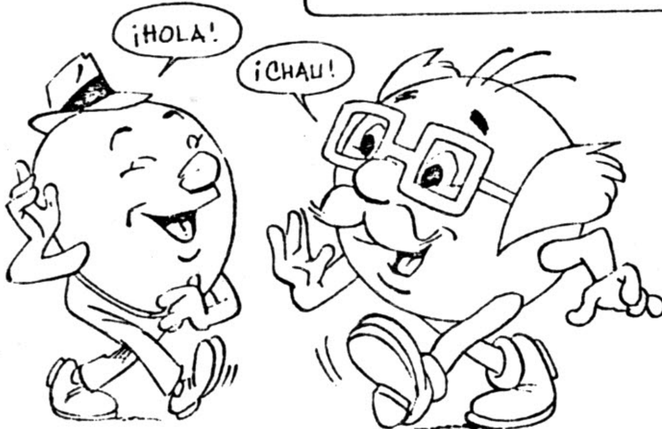
Nosotros decidimos que número de línea debe ser. Por ejemplo, podemos numerarlas 10, 20, 30... ó 100, 200, 300... ó 15, 16, 17..., como más nos guste (debemos tener en cuenta que las sentencias deben estar numeradas en el orden en que queremos que sean ejecutadas y que no podemos tener más de una línea con el mismo número).

¿Qué pasa si ingresamos una línea con el mismo número de otra que ya existe? Probemos con el programa anterior, ya tenemos la línea 20, si tipeamos y almacenamos:

```
20 STOP
```

El programa será entonces:

```
10 PRINT "HOLA"
15 PRINT "CHAU"
20 STOP
```



Para probar esto ingresemos LIST y presionemos RETURN. Notemos que la línea:

```
20 END
```

no existe más, la última línea que ingresamos está reemplazándola. Esta es una manera de modificar una línea.

## LISTADO DEL PROGRAMA

Como acabamos de ver, al ejecutarse el comando **LIST** el programa que está almacenado en la memoria aparece listado en la pantalla.

Si sólo queremos ver una determinada sección del programa, por ejemplo, desde la línea 10 a la 20 ingresemos LIST10-20 y presionamos la tecla ENTER o RETURN. Esto hace que se listen las líneas 10 y 20 y todas aquellas líneas que se encuentran entre ellas.

## BORRADO DE LINEAS

Para eliminar una línea del programa ingresamos el número de línea y presionamos la tecla ENTER o RETURN.

Si queremos eliminar un grupo de líneas contiguas podemos utilizar el comando **DELETE m-n**, donde m es el número de la primera línea a eliminar y n es el número de la última. Todas las líneas que se encuentran entre la línea m y la n son eliminadas.

## AGREGADO DE LINEAS

Para agregar una línea al programa ingresamos el número de línea, la sentencia BASIC y presionamos la tecla ENTER o RETURN.

## RENUMERACION DE LAS LINEAS

Si hemos borrado o agregado líneas, es posible que queramos renumerar las líneas de nuestro programa. Una forma sencilla de hacerlo es ingresando **RENUM** y presionando la tecla ENTER o RETURN. Las líneas del programa son automáticamente renumeradas comenzando por el número de línea 10 y con incrementos de 10, por supuesto no veremos la diferencia hasta que no hayamos ejecutado el comando **LIST**.

Podemos, también, especificar diferentes números ingresando **RENUM m, n** y presionando la tecla ENTER o RETURN (m es el número del primer número de línea y n es el valor del incremento).

Ahora que sabemos como almacenar un programa, eliminar y agregar líneas, ejecutarlo, listarlo, etc., aprendamos algunas instrucciones BASIC.

### A. REM (comentarios)

Esta instrucción nos permite incluir comentarios dentro de un programa.

A continuación de la palabra REM se escribe un mensaje textual y puede estar en cualquier lugar del programa.

La sentencia REM no es una instrucción ejecutable, pero aparece cuando listamos el programa, proporcionándonos una correcta documentación del mismo. Por ejemplo, podemos incluir al comienzo del programa una sentencia REM que nos indique que hace.

**Veamos un ejemplo:**

```
10 REM ESTE PROGRAMA CALCULA EL AREA DE UN CIRCULO
20 INPUT RADIO
30 LET AREA = 3.1416*RADIO^2
40 PRINT RADIO, AREA
50 END
```

La palabra REM puede reemplazarse por un apóstrofe ('), y el efecto es el mismo.

A veces es útil incluir en una sentencia comentarios, en ese caso el comentario debe ir al finalizar la sentencia.

Por ejemplo:

```
10 REM ESTE PROGRAMA CALCULA EL AREA DE UN CIRCULO
20 INPUT RADIO
30 LET AREA = 3.1416*RADIO^2   CALCULA EL AREA
40 PRINT RADIO, AREA ' MUESTRA EL VALOR DE RADIO Y
AREA
```

El formato de la instrucción es:

REM <comentario>

### B. LET (asignación de valores)

Esta instrucción se utiliza para asignar el valor de una expresión (es decir, valores númericos o cadenas de caracteres) a una variable.

Después de la palabra LET se cóloca el nombre de la variable, un signo igual y a continuación una constante, otra variable o una expresión aritmética.

En el proceso de asignación podemos distinguir dos etapas:

1 - Se produce un valor al evaluar una expresión y esto implica recuperar los valores de las variables de la memoria y realizar con estos valores las operaciones indicadas.

2 - Un nuevo valor se asigna a una variable, reemplazando cualquier otro valor que tuviera la variable previamente.

```
10 LET A = 4.325
20 LET B$ = "ESTOY CONTENTO"
30 LET X = 3.2*C + D^2
40 LET C$ = B$
50 LET AREA = 3.1416*RADIO^2
60 LET T1 = BASE1 + BASE2
```



La expresión del lado derecho del signo igual (=) nos da la fórmula para obtener el nuevo valor, y el nombre de la variable que va a recibir ese valor.

Por ejemplo, la línea sentencia de la línea 60 nos indica que se van a realizar las siguientes acciones:

* se recupera el valor de la variable BASE1 de la memoria
* se recupera el valor de la variable BASE2 de la memoria
* se suman los dos valores obtenidos para producir el "nuevo valor"
* se asigna el "nuevo valor" a la variable T1, reemplazando cualquier otro valor que ésta haya tenido antes.

Notemos que sólo el valor de la variable T1 cambia, ya que los valores de las variables BASE1 y BASE2 sólo son "leídos".

Aunque la expresión de una instrucción LET se parece a una expresión algebráica (de igualdad) tenemos que tener clara la diferencia.

La ejecución de la instrucción:

```
LET CUENTA = CUENTA + 1
```

hace que el valor de la variable CUENTA sea incrementado en 1.

En PECOS, no es obligatorio poner la palabra LET.

```
10 X = A*B/2 + C
20 A$ = "HOLA"
```

A veces es importante el orden en que se realizan las asignaciones. Por ejemplo:

```
60 CUENTA = CUENTA + 1
70 TOTAL = CICLOS + CUENTA
```

La ejecución de estas mismas sentencias producirá un resultado muy diferente si las escribimos en el orden inverso, o sea:

```
60 TOTAL = CICLOS + CUENTA
70 CUENTA = CUENTA + 1
```

El resultado es diferente porque una asigna un nuevo valor a CUENTA y la otra usa el valor de CUENTA. Dependiendo del orden en que las sentencias se ejecutan el nuevo valor de TOTAL reflejará el viejo o el nuevo valor de la variable CUENTA.

Debemos tener en cuenta que si bien un par de instrucciones de asignación pueden parecer un conjunto de ecuaciones, son en realidad la descripción de un proceso secuencial.

Veamos como intercambiar los valores de dos variables utilizando la instrucción de asignación. Supongamos que tenemos:

```
FILA = 16
LIMITE = 42
```

y queremos cambiar el valor de FILA por el de LIMITE y viceversa

Si hiciéramos:

```
10 FILA = LIMITE
20 LIMITE = FILA
```

no habremos logrado nuestro objetivo, ya que al ejecutarse estas dos sentencias nos quedaría:

FILA = 42

LIMITE = 42

Para intercambiar los valores de dos variables es necesario utilizar una tercer variable que conserve temporalmente uno de los valores. Lo que debemos hacer es:

```
10 TEMP = FILA
20 FILA = LIMITE
30 LIMITE = TEMP
```

y el resultado de ejecutar estas instrucciones será:

TEMP = 16

FILA = 42

LIMITE = 16

El formato general de la instrucción LET es:

[LET] <variable> = <expresión>

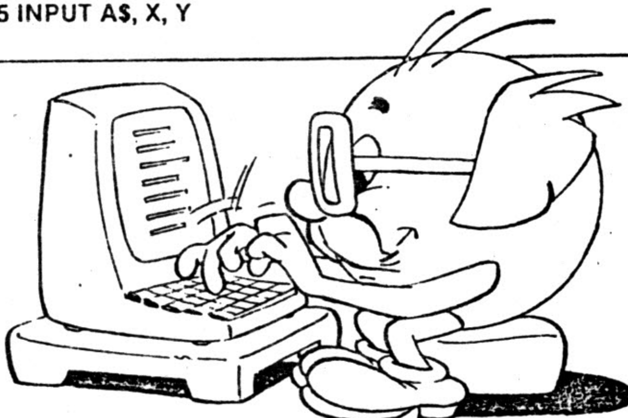
### C. INPUT (ingreso de datos)

Esta instrucción permite ingresar datos numéricos o cadenas de caracteres al computador durante la ejecución del programa.

A continuación de la palabra INPUT se escriben las variables, separadas por comas.

Por ejemplo:

```
15 INPUT X, Y, Z
25 INPUT X$
35 INPUT A$, X, Y
```



En la instrucción 15 se deberían ingresar tres valores numéricos, en la 25 una cadena de caracteres y en la 35 una cadena de caracteres y dos valores numéricos.

Al ejecutarse la instrucción INPUT aparece un signo de interrogación (?) en la pantalla, indicando que está a la espera del ingreso de los datos. La ejecución del resto del programa queda detenida, hasta recibir los datos solicitados. Una vez que los datos solicitados son ingresados éstos son asignados a cada una de las variables que se incluyen en la sentencia INPUT.

El proceso es similar al de la sentencia de asignación, en la cual se asignan a las variables valores. La diferencia es que en lugar de generar los valores mediante la evaluación de una expresión, éstos son ingresados a la computadora a través de la unidad de entrada (teclado). .

En el programita que calcula el área de un círculo usamos una instrucción INPUT, para obtener el radio. Si lo ejecutamos, cuando llega a la sentencia 2O INPUT RADIO, nos aparece un signo de interrogación y éste nos indica que está esperando que ingresemos un dato. Pero, ¿cómo sabemos qué dato está esperando? Podemos hacer que al ejecutarse la sentencia INPUT, nos muestre un mensaje antes del signo de interrogación si lo escribimos entre comillas después de la palabra INPUT.

Veamos como quedaría:

```
10 REM ESTE PROGRAMA CALCULA EL AREA DE UN CIRCULO
20 INPUT "CUAL ES EL RADIO"; RADIO
30 LET AREA = 3.1416*RADIO^2 ' CALCULA EL AREA
40 PRINT RADIO, AREA ' MUESTRA EL VALOR DE RADIO Y
AREA
50 END
```

Con esta modificación cuando nos solicite el dato aparecerá en la pantalla:

CUAL ES EL RADIO?

Los datos solicitados deben respetar las siguientes reglas:

* deben corresponder en tipo y número con las variables de la instrucción INPUT.

* pueden ser constantes numéricas o cadenas, pero no fórmulas.

El dato que es ingresado, es asignado a la variable nombrada en la instrucción.

El formato es:

INPUT [<"cadena de caracteres">] [;] <lista de variables>

Si en una sentencia INPUT incluimos una cadena de caracteres después de la palabra clave INPUT, esta cadena de caracteres aparecerá en la pantalla antes del signo de interrogación.

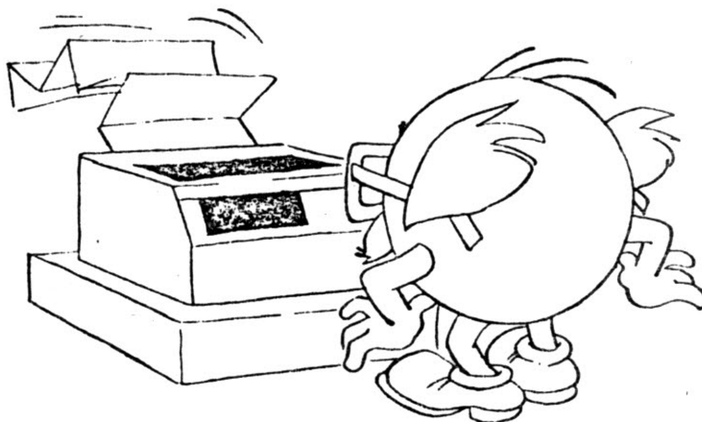
## D. PRINT (impresión de resultados)

La instrucción PRINT se utiliza para mostrar en la pantalla los resultados que nos interesan.

A continuación de la PALABRA CLAVE se pueden incluir constantes numéricas o alfanuméricas, variables o expresiones aritméticas. Estos items se pueden separar por comas (,) o punto y comas (;).

Ejemplo:

```
10 PRINT X,Y,Z
20 PRINT "X = ";X;"Y = ";Y
30 PRINT "MI NOMBRE ES ";A$
40 PRINT X,Y,(X^2 + Y^2)/2
```



La instrucción 10 hará que se impriman los valores numéricos contenidos en las variables X, Y y Z.

La instrucción 20 imprimirá X = , a continuación el valor de X y luego (en el mismo renglón) Y = y el valor de Y.

La instrucción 30 pondrá el nombre almacenado en la variable A$ a continuación de MI NOMBRE ES.

Por último, la instrucción 40 imprimirá los valores de X e Y seguidos del resultado de:

$(X^2 + Y^2)/2$

Para la escritura de esta sentencia, se deben conocer las siguientes reglas:

si los ítems que integran la lista del PRINT están separados por coma, sólo los primeros tres items van impresos en el primer renglón, los siguientes dos en el segundo, y así sucesivamente.

**Ejemplos:**

```
10 PRINT A1,A2, A3, A4, A5, A6, A7
```

Donde:

A1 = 2, A2 = 1.8, A3 = 4, A4 = 142, A5 = 5000, A6 = 8, A7 = – 6.5

Imprimirá así los valores de A1 a A7

```
–2          1.8        4
      142        50000
       8          – 6.5
```

Las comas, al separar los items de la lista, indican que cada valor debe comenzar, al ser impreso, en una de las zonas verticales en que está dividida la pantalla (PECOS divide la línea en zonas de 14 espacios cada una).

```
20 PRINT "ESTOY UN POCO", "SEPARADO"
```

Imprimirá:

ESTOY UN POCO        SEPARADO

* si se utiliza el punto y coma para separar los ítems de la lista, éstos se imprimirán uno a continuación de otro, sin espacio de separación, así:

```
10 PRINT A1; A2; A3; A4; A5; A6; A7
```

Imprimirá:

– 2..1.8..4..142..5000..8. – 6.5

Con:

```
20 PRINT "ESTOY TOTALMENTE"; "COMPRIMIDO"
```

Se obtendrá:

ESTOY TOTALMENTE COMPRIMIDO

• si la sentencia PRINT no tiene lista de variables, origina la impresión de una línea en blanco, un espaciado de renglones, así:

```
10 PRINT A1,A2,A3,A4
20 PRINT
30 PRINT A5,A6,A7
```

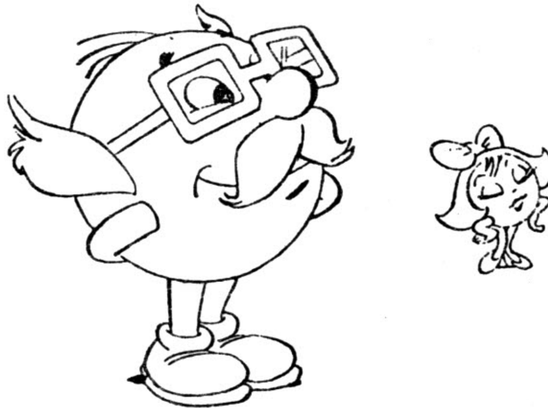
Imprimirá:

```
-2        1.8       4
                    142

.5000     8        -6.5
```

y con:

```
20 PRINT "ESTOY FATALMENTE"
30 PRINT
40 PRINT "SEPARADO"
```



Se imprimirá:

```
ESTOY FATALMENTE

SEPARADO
```

• si a continuación del último ítem de la lista de datos se coloca una coma, el siguiente dato a imprimir (primero de otra instrucción PRINT posterior) se imprimirá en el siguiente espacio disponible de la misma línea.

**Ejemplo:**

```
10 PRINT "ME PREGUNTO"; "SI ESTOY CERCA O"
20 PRINT "SEPARADO"
```



Imprimirá:

ME PREGUNTO, SI ESTOY CERCA O
SEPARADO?

```
10 PRINT A1,A2,A3,
20 PRINT A4,A5,A6,
```

Es equivalente a escribir:

```
10 PRINT A1,A2,A3;A4,A5,A6
```

Pensemos ahora que sucede en los siguientes ejemplos:

```
110 X = 3:Y = 4:Z = 5:W = 1:A = 1:B = 2
120 PRINT X,Y,Z,W
130 PRINT A,,,B
140 PRINT A;B;X;
150 PRINT Y,Z,;W
160 PRINT "COMI";X;"SANDWICHES Y";Z; "MASAS"
```

El formato general de la instrucción PRINT es:

PRINT < lista de variables >

Si ahora retomamos nuestro primer problema, que calculaba el área de un círculo, ya estamos en condiciones de hacerlo más comprensible:

```
10 REM ESTE PROGRAMA CALCULA EL AREA DE UN CIRCULO
20 INPUT "CUAL ES EL RADIO"; RADIO
30 LET AREA = 3.1416*RADIO^2 ' CALCULA EL AREA
40 PRINT "EL VALOR DEL RADIO ES"; RADIO
50 PRINT "Y EL AREA DEL CIRCULO"; AREA
60 END
```

La instrucción 10 hace que aparezca en la pantalla el pedido del radio y en el mismo renglón a continuación aparezca el signo de interrogación de la sentencia INPUT.

Esta sentencia la podríamos reemplazarla por:

```
20 PRINT "CUAL ES EL RADIO";
23 INPUT RADIO
```

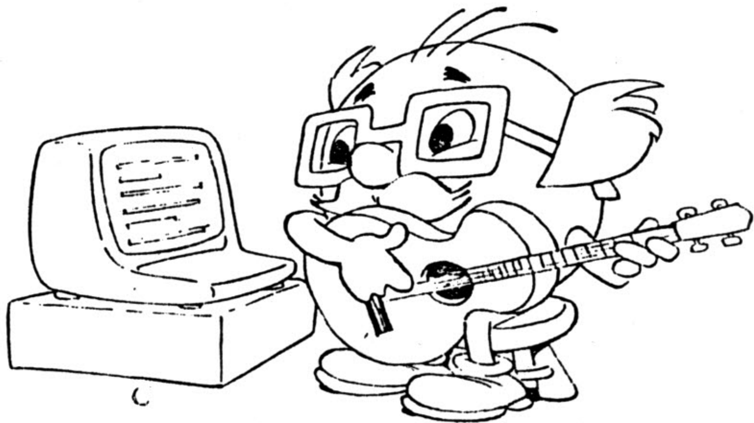
y el efecto sería el mismo, podés explicar por qué?

La instrucción 40 causa la aparición del valor del Radio, a continuación de EL VALOR DEL RADIO ES y después de 3 espacios Y EL AREA DEL CIRCULO y al lado el valor correspondiente. Te animás a explicar por qué?

Ahora sabemos lo suficiente para escribir un programa que haga ...un poema!

Ingrese esto en PECOS:

```
10 INPUT "COMO TE LLAMAS?";N$
20 PRINT "POEMA DE: "; N$
30 PRINT "**********"
40 INPUT "ESCRIBE UNA PALABRA QUE RIME CON LUNA"; X$
50 PRINT "AQUI MI POEMA"
60 PRINT
70 PRINT "CADA VEZ QUE MIRO AL CIELO Y VEO LA LUNA"
80 PRINT "ME SIENTO FELIZ COMO UNA";X$
90 END
```



Tratemos, ahora de hacer un programa que calcule las soluciones de una ecuación cuadrática.

$$X1 = \frac{-B + \sqrt{B^2 - 4AC}}{2A} \qquad X2 = \frac{-B - \sqrt{B^2 - 4AC}}{2A}$$

Asumimos que los valores de A, B y C son tales que $B^2 - 4AC$ es siempre positivo.

El algoritmo puede expresarse así:

1- leer o ingresar los valores de A, B, C

2- calcular el valor de $\sqrt{B^2 - 4AC}$

3- calcular el valor de 2A

4- obtener X1 y X2 según las fórmulas

5- imprimir A, B, C, X1 y X2

Ingresemos el programa en PECOS y analicemos los resultados (no olvidar que $\sqrt{x}$ es igual que $x^{0.5}$).

## E- IF ... THEN ... ELSE (selección de caminos)

Las instrucciones que hemos visto hasta ahora nos permiten plantear problemas cuyos algoritmos representan estructuras secuenciales: es decir, primero se ejecuta la primera instrucción, luego la segunda y así sucesivamente, hasta llegar a la última.

Sin embargo, mientras escribimos un programa, a menudo es necesario especificar dos o más cursos de acción, permitiendo, al ejecutor del programa, seleccionar uno de ellos durante la ejecución.

Esto lo podemos hacer mediante la sentencia IF ... THEN ... ELSE (si ... entonces ... sino).

Esta sentencia nos recuerda la sentencia de selección SI que ya usamos en TIMBA.

SI LA PILA A NO ESTA VACIA

TOME DE LA PILA A

SINO

TOME DE LA PILA B

NADA MAS,

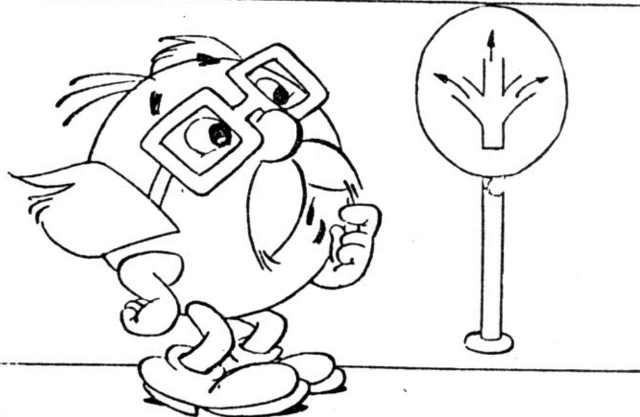
DEPOSITELA EN LA PILA C.

Recordemos que UCP verificaba si había cartas en la pila A, y de acuerdo a ello, procedía a tomar una carta de la pila A o de la pila B, si la pila A estaba vacía. En ambos casos, la carta era depositada en el tope de la pila C.

De igual forma actúa la sentencia IF de BASIC.

**Veamos un ejemplo:**

```
10 INPUT NUMERO
20 IF NUMERO >=0 THEN
     PRINT NUMERO^.5
               ELSE
     PRINT "ERROR"
30 END
```



En la línea 10 se nos pide que ingresemos un número, cuyo valor será guardado en la variable NUMERO. Una vez ingresado el procesador tendrá que seleccionar entre dos cursos de acción de acuerdo al valor contenido en la variable NUMERO. Si el valor de la variable NUMERO no es negativo, se imprimirá la raíz cuadrada del valor guardado en la variable NUMERO sino, si el valor de la variable NUMERO es negativo, se imprimirá "ERROR".

La forma general de esta sentencia es:

IF <condición> THEN <sentencia> / <Nº de línea>

ELSE <sentencia> / <Nº de línea>

Si la condición al ser evaluada es verdadera, las sentencias que están entre el THEN y el ELSE son ejectudas; si al ser evaluadas es falsa son ejecutadas las sentencias que siguen al ELSE.

Podemos omitir la segunda parte de la sentencia y escribir:

IF <condición> THEN <sentencia> / <Nº de línea>

En este caso, si la condición al ser evaluada es falsa, no se ejecutará la sentencia que sigue al THEN sino la siguiente al IF secuencialmente.

```
50 IF MENOR > MAYOR THEN MAYOR = MENOR:
            MENOR = 0
60 PRINT MAYOR
```

Si el valor de la variable MENOR es mayor que el valor de la variable MAYOR, se le asigna a la variable MAYOR el contenido de la variable MENOR y a la variable MENOR se le asigna cero.

Vean que pasa sí:

- MENOR tiene el valor 1 y MAYOR 9
- MENOR tiene el valor 200 y MAYOR -20

**Ejemplo:**

```
70 IF CARACTER$ = "DIGITO" THEN INPUT NUMERO
ELSE IF CARACTER$ = "LETRA" THEN INPUT NOMBRE$
    ELSE PRINT "ERROR"
```

Al llegar a la línea 70 se ejecutarán una de estas tres acciones, y sólo una de ellas, de acuerdo al contenido de la variable CARACTER$

* INPUT NUMERO

* INPUT NOMBRE$

* PRINT "ERROR"

Veamos más detenidamente que es una condición.

Una condición es una proposición simple o compuesta, que puede ser evaluada tanto como "VERDADERA" o "FALSA".

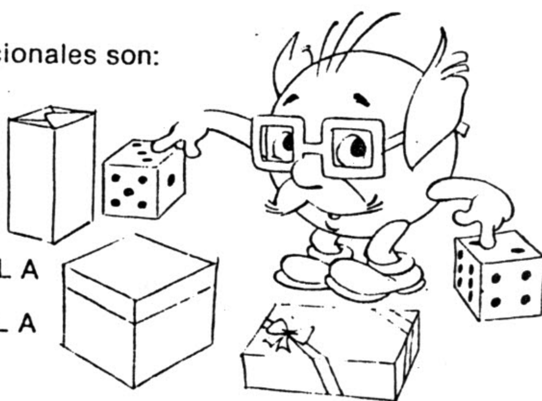
Las proposiciones son bivalentes, o sea que pueden asumir dos valores, verdadero o falso, o distinto de cero y cero. No hay confusión posible si se establece que el hecho de ser verdadera confiere a una proposición un valor distinto de cero.

No habrá tampoco confusión si se conviene en representar la verdad de una proposición por el pasaje de corriente a través de un circuito eléctrico y su falsedad por la interrupción del paso de corriente, como no la habria con cualquier otra definición, por ejemplo, si se establece que verdadero es blanco y falso es negro. El BASIC asume que FALSO es 0 y VERDADERO ES −1.

Podemos ligar proposiciones mediante operadores relacionales y/o lógicos.

Los operadores relacionales son:

| | |
|---|---|
| = | ES IGUAL A |
| < | ES MENOR QUE |
| > | ES MAYOR QUE |
| < = | ES MENOR O IGUAL A |
| > = | ES MAYOR O IGUAL A |
| < > | ES DISTINTO QUE |



Las proposiciones pueden ser no solo números o variables sino también expresiones o fórmulas.

Las siguientes líneas, por lo tanto, son válidas.

```
40 IF (X+Y)/2<>(X-Y)/2 THEN PRINT (X+Y)/2;
      "ES DISTINTO"; (X-Y)/2
50 IF (AZ^2-X)/(5*A) <=AZ+A^2 THEN A+AZ*2
60 IF XO ^ 2+5*A<25 THEN XO=25
70 IF A$="PECOS" THEN PRINT "SOY";A$
80 IF NUMERO=100 THEN PRINT "LLEGUE A 100"
      ELSE NUMERO=NUMERO+1
```

Veamos ahora los operadores lógicos.

La relación establecida por el operador lógico "Y" (AND) entre dos proposiciones en el lenguaje corriente es perfectamente clara, es decir, no da lugar a ninguna ambigüedad. Si consideramos dos proposiciones: "DARE LA ROPA VIEJA" (que llamaremos proposición A) y "DARE LA ROPA QUE ME QUEDE CHICA" (proposición B). Al decir "DARE LA ROPA VIEJA Y QUE ME QUEDE CHICA" sabemos que el "Y" agrega a la proposición A la B (decimos las dos cosas).

La relación proposicional establecida por "Y" se simboliza en lógica con el símbolo "$\wedge$". "A$\wedge$B" quiere decir "A Y B".

En BASIC la simbolizamos "A AND B".

La relación establecida entre proposiciones por el operador lógico "O" ya no es tan clara. En efecto, si analizamos un poco, veremos que en el lenguaje corriente, no tiene un significado preciso y único.

Por ejemplo, si digo "EL DOMINGO IRE AL TIGRE O A EZEIZA", para cualquiera que escuche resulta claro que si voy a un lugar no iré a otro, es decir, que una de las acciones que anuncio que realizaré excluye la otra ("ESTA TARDE ESTUDIARE O IRE AL CINE"). Si, en cambio digo "DARE LA ROPA VIEJA O QUE ME QUEDE CHICA", se entiende que, tanto la ropa vieja como la ropa que me quede chica, se incluirá en el lote de lo que daré. El "O" no es aquí excluyente.

Si en ambos casos se comprende lo que queremos decir, es por sentido general de la frase, pero desde el punto de vista lógico, es decir, cuando no nos interesamos por el contenido de las proposiciones sino exclusivamente en su valor de verdad o falsedad, es evidente que hay dos interpretaciones diferentes para la relación establecida entre proposiciones por "O".

En forma simbólica, en lógica, se adoptan dos símbolos distintos: Δ para el "O" excluyente del primer ejemplo y V para el "O" inclusivo.

Si A es "LOS RECTANGULOS ESTAN COLOREADOS y B "LAS FIGURAS EQUILATERAS ESTAN COLOREADAS", A V B significa que están coloreadas las figuras cuyos ángulos son rectos, las figuras de lados iguales y por supuesto los cuadrados que, además de ser equiláteros, tienen los ángulos rectos.

En BASIC el "O" inclusivo los representamos OR y el "O" excluyente lo representamos XOR.

Si asignamos a dos proposiciones los valores V (verdad) y F (falso) nos quedarán determinados los valores de verdad de "A AND B", "A OR B", "A XOR B", "NOT A", mediante las llamadas tablas de verdad.

| A | B | A AND B | A OR B | A XOR B | NOT A |
|---|---|---|---|---|---|
| V | V | V | V | F | F |
| V | F | F | V | V | |
| F | V | F | V | V | |
| F | F | F | F | F | V |

Si tenemos:

```
50 IF MONEDA$ = DOLARES OR MONEDA$ = "LIRAS"
   THEN PRINT "CUANTOS";MONEDA$;"VA A CAMBIAR?"
   ELSE PRINT "HA COMETIDO UN ERROR":MONEDA$ = ""
```

Como vemos el operador lógico OR liga dos condiciones y basta que una de las dos sea verdadera para que imprima el mensaje:

CUANTOS... VA A CAMBIAR?

De manera similar podríamos haber escrito:

```
50 IF MONEDA$ <>' DOLARES" AND MONEDA$ <> LIRAS"
    THEN PRINT "HA COMETIDO UN ERROR":MONEDA$= ""
    ELSE PRINT"CUANTOS":MONEDA$;"VA A CAMBIAR?"
```



Se tiene el mismo efecto pero ahora usando el operador lógico "AND". Si MONEDA$ es distinto de "DOLARES" y MONEDA$ es distinto de "LIRAS" hay un error y se imprime el mensaje "HA COMETIDO UN ERROR".

Como vemos el operador lógico AND liga dos condiciones y sólo se produce determinada acción si se cumplen las dos (ambas son verdaderas).

## F. WHILE... WEND

Para ejecutar un grupo de sentencias en forma repetida mientras se cumpla una condición dada, tenemos la sentencia WHILE... WEND (mientras... fin) que es similar a la sentencia iterativa MIENTRAS de TIMBA.

MIENTRAS LA PILA B NO ESTA VACIA

TOME DE LA PILA B

DEPOSITE LA CARTA EN LA PILA C

REPITA,

TOME DE LA PILA D

Recordemos que UCP vaciaba la pila B, pasando las cartas de a una a la pila C.

Cuando la pila B estaba vacía UCP tomaba la carta del tope de la pila D.

La forma general de la sentencia WHILE es:

WHILE <condición>

<sentencias>

WEND

Si la condición es verdadera se ejecutará el grupo de sentencias hasta encontrar el WEND, luego de lo cual la condición es nuevamente evaluada y si aún es verdadera, se repite el proceso. Si es falsa, la ejecución continúa con la próxima sentencia ejecutable después del WEND.

**Veamos un ejemplo:**




```
10 NRO = 0
20 WHILE NRO < = 10
30    NRO = NRO + 1
40    PRINT NRO
50 WEND
60 END
```

Este programa muestra los números enteros desde 1 hasta 11. Al comenzar el último ciclo la variable NRO tiene el valor 10 y al ser evaluada la condición el resultado es verdadero (10 < = 10), la línea 30 asigna a la variable NRO el valor 11 y este valor es mostrado. Al ser evaluada nuevamente la condición es falsa y el proceso finaliza.

Recordemos que la condición es evaluada solamente al comenzar cada ciclo.

Tratemos, ahora, de encontrar los 30 primeros términos de la secuencia de FIBONACCI:

1, 1, 2, 3, 5, 8, 13, 21, ......

En esta secuencia los dos primeros términos son 1. Los siguientes: son la suma de los dos términos que lo preceden. Así, el término siguiente de 21 es:

13 + 21 = 34

¿Cuál será el algoritmo para calcular los 30 primeros términos de esta secuencia.? Veamos:

```
PENULTIMO = 0

ULTIMO = 1

TERMINO = 1

Mientras TERMINO < 30

    Mostrar ULTIMO

    TERMINO = TERMINO + 1

    SUMA = ULTIMO + PENULTIMO

    PENULTIMO = ULTIMO

    ULTIMO = SUMA

Repita.
```

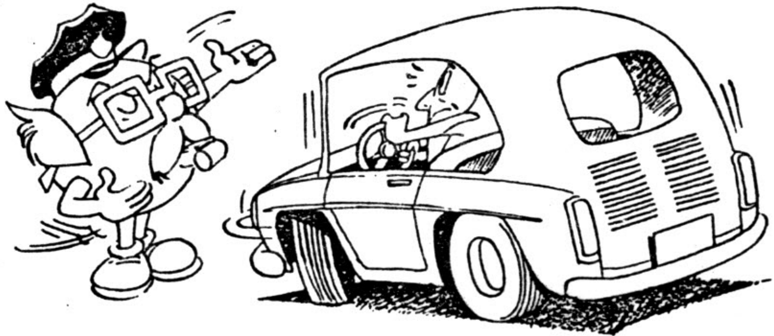
Ahora, ya podemos escribirlo en BASIC. Hagámoslo!

## G. GOTO

La sentencia GOTO indica que la ejecución debe continuarse en otra parte del programa. La forma general de una sentencia GOTO es:

GOTO < nro de línea >

Una sentencia GOTO debe reservarse solo para situaciones inusuales o poco comunes donde la estructura natural de un algoritmo debe romperse, y en general para abajo.



## EJERCICIOS

Desagregar jerárquicamente hasta completar un programa en BASIC que:

1 — Dados A > 0 y B > 0, halle el cociente entero A/B por restas sucesivas, usando una variable de cálculo auxiliar.

2 — Dados A, B y C los ordene de manera que el mayor quede en A y el menor en B.

3 — Dado N, genere e imprima los números naturales hasta N.

4 - Dado N, genere e imprima los primeros N números pares.

5 — Dados A y N, halle los N primeros múltiplos de A.

6 — Dado N, genere e imprima los números pares hasta N.

7 — Dados A y N, genere e imprima las N primeras potencias de exponente par de A.

8 — Dados A y N, genere e imprima las primeras N potencias de A, por productos sucesivos.

9 — Dados A y N, genere e imprima las potencias de exponente par de A menores o iguales que N.

10 — Dado N, genere e imprima la sumatoria de los números naturales menores o iguales que N.

11 — Dados A y N, halle la productoria de los N primeros múltiplos de A. $P = \prod_{I=1}^{N} I*A = (1*A)*(2*A)*...*(N*A)$.

12 — Dados A, B y N, genere e imprima las N primeras potencias de A siempre que estas sean menores o iguales que B.

13 — Dados A y N, genere e imprima la productoria de los N primeros números impares que no sean múltiplos de A.

14 — Dados A y B, halle los múltiplos de A menores o iguales a N.

15 — Dados A y B, halle el producto de ambos por sumas sucesivas.

16 — Dados A, B y N, halle la sumatoria de las N primeras potencias cuyas bases sean los sucesivos múltiplos de A y los exponentes los sucesivos múltiplos de B.

$$S = \sum_{I=1}^{N} (I*A)\hat{}(I*B) = (I*A)\hat{}(I*B) + (2*A)\hat{}(2*B) + \ldots (N*A)\hat{}(N*B).$$

17 — Dado N, genere e imprima la productoria de los primeros números naturales.

18 — Dados A, B y C, halle la productoria de las sucesivas potencias de A, que sean mayores que B y menores o iguales que C.

19 — Dados A, B y N, genere e imprima los primeros N múltiplos de A que no lo sean de B.

20 — Dado A, genere e imprima su factorial. A! = A*(A − 1)*(A − 2)*...*2*1.

21 — Dados A y N, genere e imprima los factoriales de los N primeros múltiplos de A.

22 — Dados A, B y C, genere la sumatoria de las B primeras potencias de A siempre que los sumandos sean menores o iguales que C.

23 — Dados A y B, halle la potencia, en que el exponente sea el menor y la base el mayor. El cálculo de la potencia no debe estar repetido, hacerlo por productos sucesivos.

24 — Dados A, B y C, genere e imprima las C primeras potencias de exponente impar de A, considerando que las potencias sean menores o iguales que B.

25 — Dados A y B, halle el producto de ambos por sumas sucesivas considerando A y B como números enteros.

26 — Dados A, B y C, genere e imprima los C primeros factoriales de los sucesivos múltiplos de A, siempre que ellos no superen a B.

27 — Dados A, B y C, halle la sumatoria de las potencias de base A cuyos exponentes son los sucesivos múltiplos de B, siempre que cada potencia sea menor que C e indique, además, el número de sumandos.

28 — Dados A, B y C, genere e imprima la productoria de los C primeros factoriales de las sucesivas potencias de A, siempre que cada factor sea menor o igual que B.

Ahora vamos a ver con más detalle los comandos que nos pueden ser de utilidad.

## NUMERANDO LAS LINEAS DEL PROGRAMA: AUTO

A menudo, resulta tedioso ingresar un programa en la memoria, ya que debemos ingresar el número de línea, la sentencia BASIC y apretar la tecla RETURN, podemos agilizar un poco nuestro trabajo mediante la orden AUTO que nos evita ingresar el número de línea.

El formato del comando AUTO es:

AUTO [<nro. de línea>[ ,<incremento>]]

Este comando genera automáticamente un número de línea cada vez que apretamos la tecla RETURN.

Comienza numerando con el<número de línea> dado, y va incrementando cada número de línea siguiente con <incremento>.Si no especificamos el<incremento>asume que es 10.

Cuando genera un número de línea que ya fue usado, después del número de línea aparece un asterisco (*) para indicarnos que cualquier sentencia que ingresemos va a reemplazar la sentencia que ya existe.

Cuando queremos que deje de generar números de líneas tenemos que apretar las teclas CONTROL y C, y esa línea no es almacenada en la memoria.

Ejemplos:

| | |
|---|---|
| AUTO 100,50 | genera los números 100, 150, 200,... |
| AUTO | genera los números 10, 20, 30, 40,... |
| AUTO ,20 | genera los números 0, 20, 40, 60,... |
| AUTO 105 | genera los números 105, 115, 125,... |

## * ORDENANDO EJECUTAR EL PROGRAMA: RUN

Si estamos listos para ejecutar el programa, podemos hacer que la computadora comience a obedecer las instrucciones que le hemos dado (PROGRAMA), dándole la orden:

```
RUN
```




La UCP comenzará a obedecer nuestro programa y, cuando llega al final del mismo nos mostrará el mensaje:

```
OK
```

y esperará más instrucciones.

El formato general del comando RUN es:

RUN [<nro. de línea>]

Si especificamos el número de línea, la ejecución comenzará en esa línea, sino comenzará en la primer línea del programa.

## • LISTANDO EL PROGRAMA: LIST

Tarde o temprano tendremos que enfrentarnos con el hecho que uno de nuestros programas, cuidadosamente elaborado, simplemente no anda, y es aquí cuando lás cosas se vuelven interesantes.

Hay ciertos errores evidentes, como por ejemplo, si no ingresamos el número de línea de cualquier línea del programa, con lo cual la computadora no aceptará esa línea; o si escribimos:

```
10 PRINT NRO
```

cuando lo ejecutamos, nos responderá con un mensaje de error:

SYNTAX ERROR IN 10

Más adelante, veremos esto con más detalle.

Supongamos que, cuando ejecutamos el programa aparece un mensaje de error y queremos saber donde nos hemos equivocado. Lo que debemos hacer es leer el programa, que está en la memoria, para encontrar la falla. Para poder leerlo, le damos a la computadora la orden:

```
LIST
```

y nos aparecerá en la pantalla todas las líneas de nuestro programa.

El comando LIST tiene dos formatos:

1. LIST [<nro de línea>]
2. LIST [<nro de línea>] – [<nro de línea>]

Veamos el formato 1:

* Si omitimos el número de línea, el programa será listado desde el principio hasta el fin , o hasta que presionemos las teclas CONTROL y C

* Si incluimos un número de línea nos mostrará sólo esa línea.

**Ejemplos:**

LIST listará todo el programa que se encuentra en la memoria.

LIST 500 listará sólo la línea número 500.



El formato 2, nos permite las siguientes opciones:

* Si sólo especificamos el primer número de línea seguido de –, esa línea y todas las que tengan un número de línea mayor serán listadas.

* Si sólo especificamos el segundo número de línea precedido de –, se listarán todas las líneas desde el comienzo del programa hasta el número de línea especificado.

* Si especificamos ambos números de línea, se listará la parte del programa que se encuentra entre esos dos números.

**Ejemplos:**

LIST 150– listará todas las líneas desde la número 150 hasta el fin.

LIST –1000 listará todas las líneas desde el comienzo hasta la 1000.

LIST 150–1000 listará desde la línea número 150 hasta la 1000 inclusive.

## * CORRIGIENDO EL ERROR: EDIT y subcomandos

Ahora bien, supongamos que hemos logrado encontrar el error, ¿cómo hacemos para corregirlo?

Podemos volver a ingresar la línea, o si no queremos ingresarla completa, podemos usar el comando EDIT.

Su formato es:

EDIT < nro de línea >.

Este comando nos permite corregir parte de una línea, sin tener que reingresarla.

Una vez que hemos ingresado el comando EDIT podemos usar subcomados que nos van a permitir mover el cursor o insertar, borrar, reemplazar o buscar determinado texto dentro de una línea.

Podemos dividir los subcomandos del EDIT de acuerdo a las siguientes funciones:

1. MOVER EL CURSOR
2. INSERTAR UN TEXTO
3. ELIMINAR UN TEXTO
4. ENCONTRAR UN TEXTO
5. REEMPLAZAR UN TEXTO
6. FINALIZAR Y RECOMENZAR LA CORRECCION



En adelante:

<car> va a representar un caracter cualquiera.

<texto> va a representar una cadena de caracteres de cualquier longitud.

<ent> va a representar un número entero, opcional.

$ va a representar la tecla ESCAPE.

## 1. MOVER EL CURSOR

Para mover el cursor hacia la derecha usamos la barra espaciadora. Si ingresamos <ent> y luego presionamos la barra espaciadora el cursor se moverá <ent> espacios hacia la derecha.

Para mover el cursor hacia la izquierda presionamos la tecla ←.

## 2. INSERTAR UN TEXTO

* Ingresando I<texto>$ podemos insertar <texto> en el lugar donde se encuentra el cursor, los caracteres que insertamos irán apareciendo en la terminal. Para terminar con la inserción, presionamos la tecla ESCAPE. Si presionamos la tecla RETURN tendremos el mismo efecto que si presionamos primero la tecla ESCAPE y luego RETURN.

Para borrar los caracteres que se encuentren a la izquierda del cursor podemos usar la tecla ←.

* Ingresando X podemos extender la línea. El cursor se mueve hasta el final de la línea y nos permite que insertemos un texto.

Para finalizar la inserción presionamos la tecla RETURN.

### 3. ELIMINAR UN TEXTO

• Ingresando <ent>D eliminamos los <ent> caracteres que se encuentran a la derecha del cursor.

Los caracteres que se eliminan aparecen en la pantalla entre barras y el cursor se ubica a la derecha del último caracter eliminado.

• Ingresando H eliminamos todos los caracteres que se encuentran a la derecha del cursor y luego podemos insertar un texto. Es útil para reemplazar textos al final de una línea.

### 4. ENCONTRAR UN TEXTO

• Ingresando <ent>S<car> podemos buscar la <ent>– sima ocurrencia del caracter <car> y posicionar el cursor antes de ella. Si el caracter <car> no es encontrado, el cursor se posiciona al final de la línea.

### 5. REEMPLAZAR UN TEXTO

• Ingresando C<car> podemos cambiar el próximo caracter por <car>. Si queremos cambiar los siguientes <ent> caracteres podemos hacerlo ingresando <ent>C<texto>.

### 6. FINALIZAR Y COMENZAR A EDITAR

• Presionando la tecla, RETURN, aparece el resto de la línea guardando todos los cambios hechos y termina la edición.

• Presionando la tecla E guarda los cambios hechos y termina la edición. La diferencia con lo anterior es que no nos muestra los cambios realizados.

• Presionando la tecla Q, terminamos con la edición sin guardar ninguno de los cambios realizados.

• Presionando la tecla L, nos muestra el resto de la línea, guarda los cambios realizados y posiciona el cursor al principio de la línea permitiéndonos continuar corrigiendo la línea.

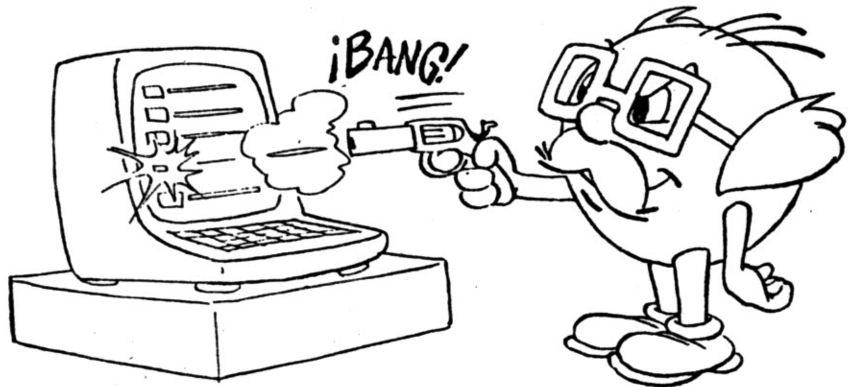
## • ELIMINANDO LINEAS DEL PROGRAMA: DELETE

Si lo que queremos hacer es eliminar determinada línea del programa debemos ingresar el número de línea y presionar la tecla RETURN o podemos utilizar el comando DELETE.

El comando DELETE tiene dos formatos:

1. DELETE <nro de línea>

2. DELETE [<nro de línea>] – [<nro de línea>]




Veamos el formato 1:

* Sólo eliminará la línea <nro de línea>

El formato 2 nos permite las siguientes opciones:

* Si sólo especificamos el primer número de línea seguido de –, se eliminarán todas las líneas desde el comienzo del programa hasta el número de línea especificado.

* Si especificamos ambos números de línea, se eliminará la parte del programa que se encuenta entre esos dos números.

Ejemplos:

DELETE 40 elimina la línea número 40

DELETE –40 elimina todas las líneas desde el comienzo hasta la 40, inclusive.

DELETE 40–100 elimina desde la línea número 40 hasta la 100, inclusive.

## * RENUMERANDO LAS LINEAS: RENUM

Si hemos borrado o agregado líneas al programa es posible que queramos renumerarlas, para lo cual utilizamos el comando RENUM.

El formato del comando RENUM es:

**RENUM** [[<nuevo nro>][,[<viejo nro>][,<incremento>]]]

Con <nuevo nro>especificamos, el primer número de línea que va a ser usado en la nueva secuencia (si no lo especificamos asume que es 10), con<viejo nro>especificamos a partir de que número de línea queremos que empiece a renumerar (si no lo especificamos asume que es a partir de la primer línea del programa)<incremento>es el incremento que se debe usar en la nueva secuencia (si no lo especificamos asume que es 10).

Tenemos que tener en cuenta que este comando no lo podemos usar para cambiar el orden de las líneas de un programa (por ejemplo, RENUM 15,30 si el programa tiene tres líneas 10, 20 y 30) o para crear números de línea (mayores que 65529.

**Ejemplos:**

**RENUM** renumera todas las líneas del programa. El número de la primer línea será 10, y el incremento también.

**RENUM 300,,50** renumera todas las líneas del programa. El número de la primer línea será 300 y el incremento será 50.

**RENUM 1000,900,20** renumera las líneas del programa a partir de la línea 900 con el número 1000 y el incremento será 20.

## * GUARDANDO EL PROGRAMA: SAVE

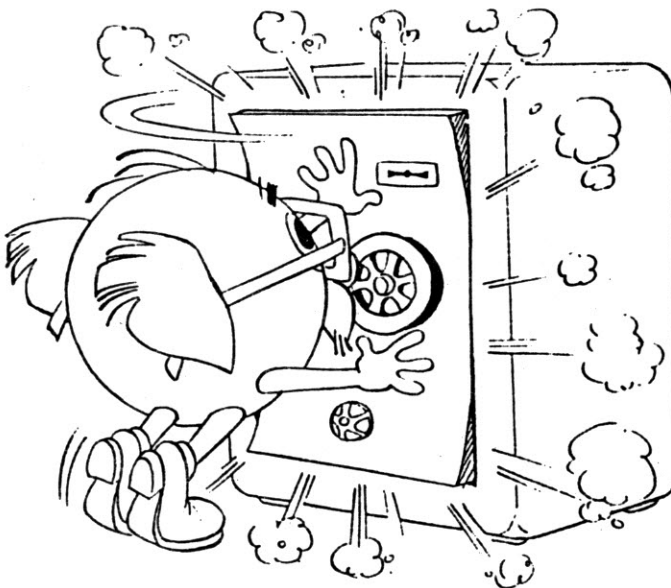
A menudo necesitamos conservar una copia de nuestro programa, ya sea para ejecutarlo luego o bien corregirlo y mejorarlo. Para guardar el programa, que tenemos en la memoria, en un diskette utilizamos el comando SAVE.

El formato de este comando es:

SAVE <nombre del programa>,A

donde <nombre del programa> debe estar encerrado por comillas.

Cuando le damos la orden de SAVE la computadora guarda el programa en el diskette y mantiene lo que tiene en la memoria.



## * ALMACENANDO EN LA MEMORIA: LOAD

Si tenemos un programa guardado en un diskette y lo queremos ejecutar, corregir o listar debemos, primero, cargarlo en la memoria. Para lo cual debemos darle a la computadora la orden de LOAD.

El formato de este comando es:

LOAD <nombre del programa>

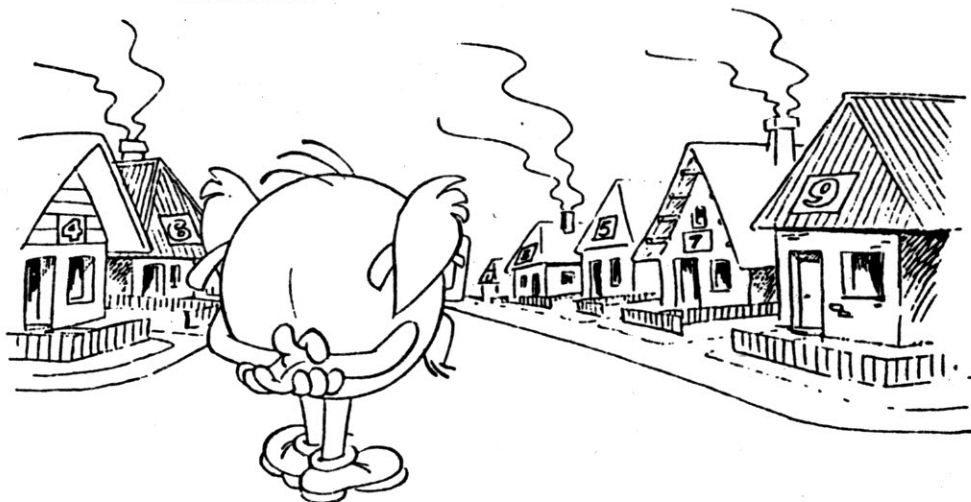
donde <nombre del programa> es el nombre con el cual lo guardamos en el diskette.



El arreglo es probablemenete la estructura de datos más difundida en los lenguajes de programación, ya que en algunos casos es la única estructura disponible explícitamente.

Un arreglo es una estructura homogénea, está compuesta por elementos de un mismo tipo, y también, se la llama estructura de acceso directo ya que todos sus componentes pueden ser seleccionados directamente y son igualmente accesibles.

Recordemos que la LISTA (unidad 6) tiene una cabeza, que se puede extraer o insertar en ella un elemento en función del contenido del mismo, independientemente de su ubicación dentro de la estructura y la LISTA no se altera si buscamos un elemento, ya sea por su ubicación o por su contenido.




Para referirnos a un componente individual se acompaña al nombre de la estructura el SUBINDICE (o SUBINDICES) que selecciona al elemento. En BASIC, el SUBINDICE debe ser un valor de tipo ENTERO, que puede tomar como mínimo el valor cero (a menos que indiquemos otro, como veremos más adelante) y que se escribe entre paréntesis.

Una variable con subíndice puede tener uno, dos, tres o más subíndices, representando respectivamente un arreglo UNI, BI, TRIDIMENSIONAL, etc. Cuando hablamos de la dimensión del arreglo nos referimos al número de subíndices y no al número de elementos; un arreglo UNIDIMENSIONAL podrá guardar muchos elementos, y un arreglo TRIDIMENSIONAL puede incluso, en principio, tener un solo elemento.

Un arreglo UNIDIMENSIONAL o VECTOR en notación matemática lo podemos escribir como $X_0, X_2, \ldots X_{19}, X_{20}$. En BASIC se escribe: **X(0), X(1), X(2), ... X(19), X(20).**

Un arreglo BIDIMENSIONAL o MATRIZ puede considerarse como compuesto por FILAS horizontales y COLUMNAS verticales. El primero de los subíndices nos indica el número de fila, empezando por cero, hasta el número total de filas y el segundo se refiere al número de columnas, empezando por cero, hasta el número total de columnas. Por ejemplo, un arreglo de dos filas por tres columnas usando la nomenclatura matemática será:

$A_{00}$ $A_{01}$ $A_{02}$
$A_{10}$ $A_{11}$ $A_{12}$

En BASIC, **los elementos los escribimos:**

**A(0,0), A(0,1), A(0,2), A(1,0), A(1,1), A(1,2).**

Podemos ver que los subíndices los separamos por medio de comas.

Veamos esto con un ejemplo, supongamos que tenemos un mazo de cartas ordenado de acuerdo a su PALO y a su VALOR.



| PALO | VALOR DE LA CARTA | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | (0) | (1) | (2) | (3) | (4) | (5) | (6) | (7) | (8) | (9) |
| (0) BASTOS | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 10 | 11 | 12 |
| (1) OROS | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 10 | 11 | 12 |
| (2) ESPADAS | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 10 | 11 | 12 |
| (3) COPAS | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 10 | 11 | 12 |

Consideramos el PALO de la carta como una dimensión y al VALOR de la carta como la segunda dimensión. Este es por lo tanto un arreglo de dos dimensiones.

La numeración de las FILAS y las COLUMNAS nos permiten localizar un elemento individual de un arreglo. Por ejemplo: el UNO de ESPADAS, está en la FILA 2 y en la columna 0, usando los subíndices: (2,0). De la misma manera el CABALLO de COPAS está en la posición (3,8). Notemos que el subíndice de la FILA siempre se especifica antes que el subíndice de la COLUMNA.

Podemos considerar un arreglo de tres dimensiones como formado por PLANOS, cada uno de los cuales está formado por FILAS y COLUMNAS.

Como dijimos los arreglos son conjuntos de elementos del mismo tipo. En BASIC, pueden tener como máximo 255 dimensiones y cada dimensión puede tener como máximo 32767 elementos.

Este tipo de estructura se puede usar para manejar datos relacionados, como ser una tabla con las edades de varias personas, números de teléfonos, valores de una divisa a través de los meses, etc.

Un arreglo puede estar formado por constantes numéricas o alfanuméricas, pero todos los elementos deben ser del mismo tipo.

Para nombrar un arreglo usamos las mismas reglas o convenciones que usábamos con las variables sin subíndices.

Supongamos que tenemos dos puntos en el espacio, representados en su forma cartesiana por sus coordenadas $X_1$, $X_2$, $X_3$ e $Y_1$, $Y_2$, $Y_3$, y queremos calcular la DISTANCIA entre ellos, dada por la fórmula:

$$\text{DISTANCIA} = \sqrt{(X_1 - Y_1)^2 + (X_2 - Y_2)^2 + (X_3 - Y_3)^2}$$

Ahora, supongamos que hemos establecido un arreglo llamado X cuyos tres elementos son las coordenadas del punto X y otro similar para los del punto Y.

El cálculo de la DISTANCIA entre los puntos la podemos obtener mediante la siguiente expresión en BASIC.

```
DISTANCIA =
[(X(1) – Y(1))^2 + (X(2) – Y(2)) ^2 + (X(3) – Y(3)^2)] ^0.5
```

Una variable con subíndice es raro que aparezca en un programa con una constante entera como subíndice. Generalmente, el subíndice se escribe con el nombre de otra variable (una cuyo valor sea un número entero).

Por ejemplo, consideremos un arreglo llamado CUENTA. Podemos referirnos a un elemento del arreglo escribiendo CUENTA(I). El valor de la variable I en el momento que la referenciemos puede ser 1, 2, 3 ó 4. El valor de la variable I especifica a cual del conjunto de variables llamadas CUENTA nos estamos refiriendo. Esto permite que una variable se use para seleccionar un elemento particular de un arreglo, o diciéndolo de otra forma, asignándole un valor a la variable I el programa puede determinar cual elemento del arreglo CUENTA va a ser procesado.

El valor de un subíndice puede estar dado, también, por una expresión, o sea, una fórmula que contenga variables, constantes y operadores aritméticos y con la cual pueda obtenerse un valor entero. Por ejemplo, los valores de los subíndices pueden ser especificados de la siguiente forma:

```
CUENTA (I + 2)
CUENTA (I + J)
CUENTA (K + LIMITE/2)
```

Una expresión es válida como subíndice si al ser evaluada produce un valor entero que, corresponda a uno de los elementos del arreglo.

Los subíndices pueden ser a su vez variables o expresiones. Esto significa que podemos realizar cálculos básicos, y efectuar después los mismos cálculos con diferentes valores, simplemente cambiando el valor de los subíndices.

Supongamos, que necesitamos calcular la suma de los cuadrados de 20 números desde $X_0$ hasta $X_{19}$ que se encuentran almacenados en la computadora. Podemos por supuesto asignarles 20 nombres distintos y establecer así una larga expresión aritmética para calcular la suma de sus cuadrados, pero esto sería tedioso, engorroso e inflexible. En su lugar establecemos que los números son elementos de un arreglo UNIDIMENSIONAL al que llamamos NUMERO. De esta forma, podemos referirnos a cualquiera de los 20 escribiendo NUMERO(I), sujetándonos después a las normas para que I tome todos los valores desde 0 hasta 19.

Esto se puede efectuar con el siguiente programa:

```
10 SUMA = 0
20 I = 0
30 WHILE I < = 19
40    SUMA = SUMA + NUMERO(I)^2
50    I = I + 1
60 WEND
70 END
```

Resumiendo:

Cada elemento de un arreglo puede ser accedido usando subíndices. Los subíndices son variables , constantes o expresiones que se colocan encerradas entre paréntesis. A(2) se refiere al tercer elemento de un vector llamado A, y MAT(3,2) al elemento ubicado en la cuarta fila y tercer columna de la matriz MAT.

## 2. SENTENCIA DIM

Cuando en un programa utilizamos arreglos cuyos subíndices pueden tomar valores mayores que 10 tenemos que proporcionar cierta información al respecto:




* ¿qué variables tienen subíndices?

* ¿cuántos subíndices tiene cada uno de los arreglos?

* ¿cuántos elementos existen en cada arreglo, o sea, cuál es el valor máximo de cada subíndice?

Esta información la proporcionamos mediante la sentencia DIM.

Cada una de las variables con subíndice dentro de un programa deberán mencionarse en una sentencia DIM si los subíndices pueden tomar valores mayores que 10 y esta sentencia deberá escribirse antes que algunos de los elementos del arreglo sea mencionado.

Si escribimos:

```
10 DIM UNO(15)
```

Estaremos declarando un arreglo llamado UNO, que es UNIDIMENSIONAL y cuyo subíndice puede tomar como máximo el valor 15.

Si escribimos:

```
20 DIM DOS(12,13)
```

Estaremos declarando un arreglo llamado DOS, BIDIMENSIONAL, y que tendrá como máximo 13 filas y 14 columnas.

Al dimensionar un arreglo definimos su capacidad máxima.

En una sentencia DIM podemos incluir varias variables con subíndice, por ejemplo:

```
10 DIM X(20),NUMERO(3,9),PALABRA$(30)
```

Esta sentencia asignará 21 espacios al arreglo UNIDIMENSIONAL X, 40 espacios al arreglo BIDIMENSIONAL NUMERO (4 x 10) y 31 espacios al arreglo UNIDIMENSIONAL PALABRA$ cuyos elementos son de tipo STRING.

El máximo número de dimensiones que puede especificarse en una sentencia DIM es 255 y los arreglos deben ser declarados sólo una vez en el programa, es decir, que arreglos de diferentes dimensiones deben tener diferente nombre.

La sentencia DIM es de la forma:

DIM<Item>[,<Item>]

donde <Item> es:

<variable> ( <número> [,<número>])

y<número>es el valor máximo que puede tomar el subíndice.

Trataremos de hacer un programa que calcule los 1000 primeros números primos. Recordemos para esto el algoritmo que vimos en la unidad 6.

Comienzo

```
    Definir los enteros K, J, RESTO, COCIENTE, POSPRI
    K ←— 2
    POSPRI ←— 3, PRIMOS(1) ←— 2, PRIMOS(2) ←— 3
    Mientras K < 1000
            J ←— 1
            RESTO ←— 1
            POSPRI ←— POSPRI + 2
            Mientras RESTO < > 0 y J < K
                J ←— J + 1
                COCIENTE ←— POSPRI + 2
                RESTO ←— POSPRI – (PRIMOS(J)*COCIENTE)
            repita
            Si  RESTO < > 0
                K ←— K + 1
                PRIMOS(K) ←— POSPRI
            sino nada más
    repita
fin
```

Recordemos que:

PRIMOS: es la lista donde se almacenan los primeros mil números primos.

K: es una variable que tomará los valores de 1 a 1000 y que servirá, cada vez, para indicar en que posición de la lista hay que almacenar el primo identificado a su turno.

J: es una variable que, para cada valor POSPRI, irá tomando sucesivamente el valor de las posiciones de los primos ya almacenados.



RESTO: es una variable que tiene el valor del resto de la división del posible primo por los primos menores que él.

COCIENTE: es una variable que tiene el valor del cociente de la división del posible primo por los primos menores que él.

POSPRI: es una variable que tiene el valor del número candidato a determinar si es primo.

En BASIC esto sería:

```
10 DIM PRIMOS(1000)
20 K=2
30 POSPRI=3
40 PRIMOS(1)=2
50 PRIMOS(2)=3
60 WHILE K <1000
70      J=1
80      RESTO=1
90      POSPRI=POSPRI+2
100     WHILE RESTO < >AND J < K
110          J=J+1
120          COCIENTE=POSPRI/PRIMOS(J)
130          RESTO=POSPRI-(PRIMOS(J)*COCIENTE)
140     WEND
150     IF RESTO<>0 THEN K=K+1:PRIMOS(K)=POSPRI
160 WEND
170 END
```

## 3. SENTENCIA READ Y SENTENCIA DATA

Los valores de un arreglo pueden asignarse dentro del mismo programa o pueden ser ingresados a través del teclado.

Para ingresarlos a través del teclado utilizamos la sentencia INPUT y para asignarlos dentro del programa podemos usar la sentencia LET. Estas sentencias las vimos en la unidad 7. Si se trata de una gran cantidad de datos, puede ser muy engorroso. Para facilitar la tarea existen dos sentencias relacionadas: READ y DATA, que pueden reemplazar tanto a la sentencia INPUT como a la sentencia LET y que veremos a continuación.




La sentencia DATA provee los valores a ser asignados a las variables. Mientras que la sentencia READ especifica las variables para las cuales se deben obtener valores de la sentencia DATA.

El formato de la sentencia DATA es:

DATA < lista de constantes >

La sentencia DATA es una sentencia no ejecutable y puede colocarse en cualquier lugar del programa. Puede contener tantas constantes como entren en una línea, y éstas deben estar separadas por comas. La lista de constantes puede contener tanto constantes numéricas como alfanuméricas. Las constantes alfanuméricas deben estar encerradas entre comillas si tienen comas, dos puntos o blancos significativos, de otra manera no es necesario.

El formato de la sentencia READ es

READ < lista de variables >

Una sentencia READ debe usarse siempre junto con una sentencia DATA. Esta sentencia asigna a las variables los valores de la sentencia DATA uno a uno. Las variables pueden ser numéricas o alfanuméricas y el valor leído debe estar de acuerdo con el tipo de variable especificado. Una sentencia READ puede acceder a una o más sentencias DATA (pero serán accedidas en orden) o varias sentencias READ pueden acceder a una misma sentencia DATA.

Si el número de variables excede el número de constantes de las sentencias DATA nos aparecerá durante la ejecución un mensaje de error.

**Ejemplos:**

```
10 DIM TOT(3)
20 DATA 88,77,ENERO,200,10,12,15
30 READ A,B,MES$,DINE$,TOT(1),TOT(2),TOT(3)
40 PRINT "A = ";A;"B = ";B
50 PRINT "MES = ";MES$;"DINERO = "DINE$
60 PRINT "TOTALES PARCIALES:";TOT(1)
70 PRINT "                    ";TOT(2)
80 PRINT "                    ";TOT(3)
90 END
```

Este programa imprimirá:

```
A = 88 B = 77
MES = ENERO DINERO = 200
TOTALES PARCIALES = 10
                    12
                    15
```

## 4. SENTENCIA FOR ... NEXT

Cuando queremos que un grupo de sentencias se ejecute en forma repetida, un determinado número de veces, prefijado de antemano, la sentencia apropiada es FOR ... NEXT.

La forma general de esta sentencia es:

FOR <variable> = <x>TO <y>[ STEP <z>]

<sentencias>

NEXT<variable>

donde x, y, z son expresiones numéricas.




La variable se usa como contador. La primera expresión numérica <x> es el valor inicial del contador. La segunda expresión numérica <y> es el valor final del contador.

Las sentencias son ejecutadas hasta que la palabra NEXT es encontrada. Luego, el contador se incrementa por la cantidad especificada en STEP, o sea, la tercer expresión numérica <z>. Se realiza una verificación para ver si el valor del contador es mayor que el valor final <y>. Si no lo es, se vuelve a ejecutar el grupo de sentencias contenidas en el FOR y el proceso se repite. Si la parte de STEP no se especifica, se asume que el incremento es 1. Si el incremento es negativo, el valor final del contador debe ser menor que el valor inicial del mismo. El contador se decrementa cada vez que se llega al NEXT y el grupo de sentencias se ejecuta hasta que el contador es menor que el valor final.

Ejemplo:

```
10 INPUT NROTERMINO
20 SUMA = 0
30 FOR TERMINO = 1 TO NROTERMINO
40    SUMA = SUMA + 1/TERMINO
50    PRINT SUMA
60 NEXT TERMINO
70 END
```

El efecto de la sentencia FOR en este programa es el de ejecutar la asignación

```
SUMA = SUMA + 1/TERMINO
```

Para cada valor entero de la variable TERMINO desde TERMINO = 1 hasta TERMINO = NROTERMINO.

Si NROTERMINO es 1 entonces la asignación no será ejecutada.

Supongamos que deseamos imprimir los cuadrados de todos los números naturales comprendidos entre 5 y 12, ambos incluídos. Aquí la acción que hay que repetir 8 veces es "elevar un número al cuadrado e imprimir el resultado". Esto hay que hacerlo con el número 5, luego con 6 y así sucesivamente hasta llagar a hacerlo con 12. En este ejemplo el número es incrementado de uno en uno. Una forma de resolver esto es:

```
10 NRO = 5
20 WHILE NRO < = 12
30      CUADRADO = NRO^2
40      PRINT NRO,CUADRADO
50      NRO = NRO + 1
60 WEND
70 END
```

¿Cómo lo resolveríamos con la sentencia FOR?

Si quisiéramos imprimir los cuadrados de los números naturales pares comprendidos entre 1 y 29, ¿cómo la haríamos?

Dentro de una sentencia FOR podemos tener otras sentencias FOR.

**Ejemplo:**

```
10 PRINT "I","J"
20 FOR I = 1 TO 4
30      FOR J = 1 TO 6
40           PRINT I,J
50      NEXT J
60 NEXT I
70 END
```

Cuando hay sentencias FOR anidadas, cada una debe tener una variable distinta como contador. La sentencia NEXT, para el lazo más interno, debe aparecer antes que la del lazo externo.

```
FOR JI = 1 TO 10
     FOR J2 = 2 TO  8
     NEXT J2
NEXT J1
```

Esto es correcto

```
FOR J1 = 1 TO 10
     FOR J2 = 2 TO  8
NEXT J1
     NEXT J2
```

Esto es incorrecto

Lo más difícil del proceso de desarrollar un programa es sin duda la planificación y el diseño del algoritmo y no la "traducción" de dicho diseño a las sentencias de un lenguaje de programación. Por ejemplo, es más fácil aprender los "movimientos" de las distintas piezas de ajedrez que aprender como usar estos movimientos para jugar una buen partida.

Por eso, insistimos en que lo más importante es manejar la metodología vista en la unidad 5 del tomo 1.

Veamos ahora, dos ejemplos completos, dado un problema llegar a un programa BASIC.

## Ejemplo1:

Dada una lista de números, listar los números (uno por línea) en el orden dado, pero terminando de listar cuando aparezca el mayor número de toda la lista.

Por ejemplo, suponiendo que la lista dada sea: 1, 7, 3, 9, 5, 0 se listarán los números:

1

7

3

9

La "trampa" es que no sabemos cuando terminar de listarlos hasta que no determinemos cual es el mayor de la lista, y no sabemos cuál es el mayor de la lista hasta que no examinemos la lista completa.

Es obvio, que la lectura debe estar separada de la impresión, el problema lo podemos dividir en:

1 - leer la lista
2 - encontrar el mayor valor
3 - listar desde el primero hasta el mayor

Es importante que nos demos cuenta que sólo podemos leer los dato una vez. O sea, el programa no puede leer todos los números de la lista para determinar cuál es el mayor y luego, volverla a leer para listar. Por lo tanto, el programa debe almacenar la lista completa en la memoria, de manera que los valores, estén disponibles para ser listados. Esto requiere una estructura de datos que almacene la lista de números, obviamente un arreglo. Supongamos que llamamos al arreglo LISTANRO, ¿de qué tipo serán los elementos de LISTANRO? En el ejemplo, usamos sólo valores enteros, pero en la descripción del problema no hay nada que nos diga que deben ser solo enteros, entonces serán números reales.¿Cuántos elementos tendrá LISTANRO? Supongamos que 80 elementos.

Ahora podemos reescribir nuestro problema en función de LISTANRO:

1 - leer la lista y guardarla en LISTANRO
2 - encontrar en LISTANRO el mayor valor
3 - listar los valores de LISTANRO, desde el primero hasta el de valor mayor.

Tengamos en cuenta que si bien LISTANRO puede tener 80 elementos, en general, no necesariamente todos serán usados. Entonces, necesitamos una variable que nos diga cuántos elementos de LISTANRO son usados y la llamaremos LONGLISTA. Necesitamos otra variable que guarde la posición del mayor valor POSCMAYOR. Nos queda entonces:

1 - obtener LONGLISTA, leer y guardar LISTANRO.
2 - encontrar el mayor valor de LISTANRO, desde el elemento 1 hasta LONGLISTA, y guardar su posición en POSCMAYOR.
3 - listar los valores de LISTANRO, desde 1 hasta POSCMAYOR.

Los pasos 1 y 2 los podemos combinar y hacer un ciclo que lea y chequee si es el mayor valor. Entonces el algoritmo será:

1 - almacenar LISTANRO y guardar en POSCMAYOR la posición del mayor valor.
2 - listar LISTANRO, desde 1 hasta POSCMAYOR.

El primer problema lo resolvemos con un ciclo que almacene los números leídos en el siguiente elemento de LISTANRO y determine la posición del mayor valor.

```
Leer LONGLISTA
MAYORNRO ←— – 1E20
Mientras I < = LONGLISTA
     Leer LISTANRO(I)
     Si LISTANRO(I) > MAYORNRO entonces
          POSCMAYOR ←— I
          MAYORNRO ←—LISTANRO(I)
     sino nada más
     I ←—I + 1
repita
```

En BASIC esto sería:

```
INPUT LONGLISTA
MAYORNRO = – 1E20
FOR I = 1 TO LONGLISTA
     INPUT LISTANRO(I)
     IF LISTANRO(I) > MAYORNRO THEN
          POSCMAYOR = I:MAYORNRO = LISTANRO(I)
NEXT I
```

Una vez que conocemos la posición del mayor número de la lista podemos imprimirla:

Mientras I < = POSCMAYOR
    Imprimir LISTANRO(I)
    I ⟵ I + 1

repita

En BASIC será:

```
FOR I = 1 TO POSCMAYOR
        PRINT LISTANRO(I)
NEXT I
```

Uniendo todos los pedazos nos quedará el siguiente programa BASIC:

```
 10 INPUT LONGLISTA
 20 MAYORNRO = - 1E20
 30 FOR I = 1 TO LONGLISTA
 40      INPUT LISTANRO(I)
 50      IF LISTANRO(I) >MAYORNRO THEN
            POSCMAYOR = I:MAYORNRO = LISTANRO(I)
 60 NEXT I
 70 FOR I = 1 TO POSCMAYOR
 80      PRINT LISTANRO(I)
 90 NEXT I
100 END
```

**Ejemplo 2:**

Sea una "lista" de números y un conjunto de "demandas" (números que pueden estar o no en la lista). Para cada demanda determinar si está o no en la lista. Si está, indicar que posición ocupa, si no está explicar el hecho.

Por ejemplo, con la lista (9, – 4.5, 16, 5) diremos que la demanda "16" está en la posición 3, las demandas "12" y "4.5" no están en la lista.

Para aclarar el problema, detallemos en que consiste la "entrada" del mismo.

Formalmente tendremos:

a) un entero que especifica la longitud de la lista
b) la lista de valores
c) las demandas

La lista tendrá a lo sumo 100 valores reales. En el ejemplo sería:

a) 4
b) 9, – 4.5, 16, 5
c) 16, 12, 4.5

Veamos ahora la salida.

Para cada demanda imprimir una línea con el valor y su posición en la lista o una leyenda que diga que no está en la lista.

En el ejemplo sería:

16 está en la posición 3
12 no está en la lista
4.5 no está en la lista



Ahora, consideremos los posibles errores que pueden darse en los datos de entrada. Por ejemplo, el primer valor, que especifica la longitud de la lista, puede ingresarse equivocadamente. Hay que decidir que verificaciones hacer y que acción tomar en cada caso. Entonces podemos expresar más concretamente: la longitud de la lista debe ser un entero de 0 a 100, sino, terminar la ejecución dando el correspondiente mensaje de error. El número de demandas es arbitraria.

Supongamos que hay un error en los valores de la lista y se ingresa el número 9 en lugar del 8. No hay manera en el programa de descubrir este error. Podríamos convenir que los números estuvieron acotados (no fueron mayores que un determinado número) y por lo tanto, rechazaríamos cualquier valor menor, pero esto no ayuda mucho. Pero consideremos, sólo para visualizar el tratamiento de errores en los datos, la siguiente restricción en nuestro problema: los valores de la lista estarán comprendidos entre – 30 y + 30; cualquier valor fuera de este rango debe ser rechazado con un mensaje apropiado.

Ahora, consideremos la estrategia de la solución. En principio existen 2 listas totalmente separadas. Es decir, la estructura principal del algoritmo constará de 2 pasos diferenciados:

V0:

1 - cargar la "lista" completa en la memoria
2 - procesar las "demandas", una por vez

Esta es la versión 0 de nuestro programa. Noten, que en este nivel tenemos un algoritmo estrictamente secuencial, se realiza el paso 1, luego el paso 2 y el proceso termina. Cada uno involucra internamente un ciclo, pero en este nivel aún no lo explicitamos.

Ahora, tenemos dos problemas simples:

V1:

PROBLEMA 1: dado un número seguido por una lista de números, almacenar la lista en la memoria.

PROBLEMA 2: dada una lista de números en la memoria y otra lista de datos; informar la posición en la lista almacenada de cada valor de la lista de datos.

La única relación entre estos dos problemas es la lista de números en la memoria. El problema 1 produce la lista, el problema 2 usa la lista. Estos problemas pueden analizarse independientemente.

Veamos, primero, que especificaciones tiene la lista. Tendrá una longitud y un conjunto de valores. Dichos valores serán números y reales. La longitud es a lo sumo 100, entonces creamos un vector de 100 elementos. Elijamos el nombre NROLISTA para este arreglo. Necesitaremos, también una variable para el valor de la longitud de la lista, llamámosla LONGLISTA (puede ser entero).

Por ahora, con estas estructuras de datos, podemos empezar a precisar los problemas 1 y 2. Por supuesto necesitaremos otras variables, pero las iremos precisando a medida que desarrollemos el algoritmo.

Reescribamos los problemas 1 y 2 en términos de la decisión que tomamos sobre la estructura de datos.

V2:

PROBLEMA 1: almacenar LONGLISTA y NROLISTA(1 a LONGLISTA)

PROBLEMA 2: para cada demanda, informar su posición en NROLISTA(1 a LONGLISTA).

Leer LONGLISTA

Leer NROLISTA(I), variando I de 1 a LONGLISTA con incremento de 1.

Necesitamos entonces una variable I.

Tanto LONGLISTA como los valores de la primera lista deben estar en un determinado rango. Las sentencias para hacer esto deben tener la siguiente forma:

Si el valor es impropio entonces

Tomar la acción apropiada

sino nada más

En el caso de LONGLISTA, la condición debe detectar valores fuera del rango (1:100) y la acción apropiada es emitir un mensaje de error y terminar el programa.



```
Si LONGLISTA <1 o LONGLISTA >100 entonces
     Imprima "LONGITUD IMPROPIA DE LA LISTA"
          "VALOR DADO ES:", LONGLISTA
     Ir a fin de programa
sino nada más
```

Elegimos un mensaje que sea claro para el usuario, teniendo en cuenta que él no sabe como funciona el programa (es decir, seria confuso imprimir un mensaje que dijera: "LONGLISTA = 200").

Nótese que debemos usar una sentencia que fuerce el salto incondicional al fin del programa.

Para verificar los elementos de NROLISTA la condición es simple:

NROLISTA < – 30 O NROLISTA > 100

Lo que hay que decidir es cual es la acción apropiada. La especificación dice que cualquier valor fuera del rango debe ser rechazado, es decir no debe ir a NROLISTA. Es decir, que la lista interna o sea la almacenada en memoria puede resultar de longitud menor que la que tenemos en los datos y LONGLISTA no puede reflejar longitud de las dos. Por lo tanto, necesitamos otra variable entera, llamémosla LONGLISTINT.

Entonces, resultaría:

```
Leer LONGLISTA
Si LONGLISTA <1 ó LONGLISTA >100 entonces
     Imprima "LONGITUD IMPROPIA DE LA LISTA"
               "VALOR DADO ES:",LONGLISTA
     Ir a fin de programa
sino nada más
Mientras I <LONGLISTA
     Leer número
     Si número <– 30 ó número >30 entonces
          Imprima "VALOR IMPROPIO:";número
     sino
          LONGLISTINT = LONGLISTINT + 1
          NROLISTA  (LONGLISTINT) = número
     nada más
repita
```

Nosotros sabemos en realidad, cuantos elementos debería tener la lista, por lo tanto, en BASIC, podremos usar la sentencia FOR ... NEXT.

```
DIM NROLISTA(100)
LONGLISTINT = 0
INPUT LONGLISTA
IF LONGLISTA < 1 OR LONGLISTA > 100 THEN
    PRINT "LONGITUD IMPROPIA DE LA LISTA":
    PRINT "VALOR DADO ES:";LONGLISTA:
    GOTO fin de programa
FOR I = 1 TO LONGLISTA
    INPUT NUMERO
    IF NUMERO < – 30 OR NUMERO > 30
        THEN PRINT "VALOR IMPROPIO:";NUMERO
        ELSE LONGLISTINT = LONGLISTINT + 1:
            NROLISTA(LONGLISTINT) = NUMERO
NEXT I
```

Ahora, consideremos el problema 2, el de procesamiento. Primero observen que cada demanda es independiente de la otra. Las demandas pueden leerse y procesarse una por vez, y sólo una de ellas tiene que estar en la memoria en un determinado momento.

El problema sería:

Mientras haya demandas
    Leer demanda
    Procesarla
repita



Debemos definir la tarea "procesar una demanda". Veremos como hacerlo con una en particular y luego incorporaremos el proceso al ciclo que analiza todos los datos buscados.

Necesitamos, obviamente un ciclo para comparar cada elemento de NROLISTA con la demanda:

```
I = 1, ENCONTRE = 0
Mientras I < = LONGLISTINT y NO ENCONTRE
     Si NROLISTA(I) = DEMANDA entonces
          POSICION = I
          ENCONTRE = − 1
     sino
          I = I + 1
     nada más
     si no ENCONTRE entonces
          Imprima DEMANDA "NO ESTA EN LA LISTA"
     sino
          Imprima demanda "ESTA EN LA POSICION",I
     nada más
repita
```

Fíjense que tenemos que indicar de alguna manera si tiene éxito la búsqueda o no para decidir que imprimir.

En BASIC sería:

```
I = 1:ENCONTRE = − 1
WHILE I < = LONGLISTINT AND NOT(ENCONTRE)
     IF NROLISTA(I) = DEMANDA
          THEN POSICION = I:
               ENCONTRE = 0
          ELSE I = I + 1
WEND
IF NOT(ENCONTRE)
     THEN PRINT DEMANDA;"NO ESTA EN LA LISTA"
     ELSE PRINT DEMANDA;"ESTA EN LA POSICION";I
```

Ahora, tenemos que elegir como determinamos que no hay más demandas que procesar. Una solución sería preguntar cada vez si no hay demandas:

```
Imprimir "QUIERE BUSCAR UN NUMERO EN LA LISTA?"
Leer respuesta
Mientras respuesta = "SI"
    Leer demanda
    Procesar demanda
    Imprimir "QUIERE BUSCAR UN NUMERO EN LA
            LISTA?"
    Leer respuesta
repetir
```

En BASIC:

```
INPUT "QUIERE BUSCAR UN NUMERO EN LA
        LISTA";RESPUESTA$
WHILE RESPUESTA$ = "SI"
     INPUT DEMANDA
     I = 1:ENCONTRE = -1
     WHILE I < = LONGLISTINT AND NOT(ENCONTRE)
          IF NROLISTA(I) = DEMANDA
               THEN POSICION = I:ENCONTRE = 0
               ELSE I = I + 1
     WEND
     IF NOT(ENCONTRE)
          THEN PRINT DEMANDA;"NO ESTA EN LA LISTA"
          ELSE PRINT DEMANDA;"ESTA EN LA POSICION";I
          INPUT "QUIERE BUSCAR UN NUMERO EN LA
                  LISTA";RESPUESTA$
WEND
```

Juntando todas las piezas obtenemos el siguiente programa en BASIC:

```
10 DIM NROLISTA(100)
20 LONGLISTINT = 0
30 INPUT LONGLISTA
40 IF LONGLISTA <1 OR LONGLISTA >100 THEN
        PRINT "LONGITUD INAPROPIADA DE LA LISTA":
        PRINT "VALOR DADO ES:";LONGLISTA:
        GOTO 190
50 FOR I = 1 TO LONGLISTA
60    INPUT NUMERO
70    IF NUMERO <- 30 OR NUMERO >30
        THEN PRINT "VALOR IMPROPIO:";NUMERO
        ELSE LONGLISTINT = LONGLISTINT + 1:
              NROLISTA(LONGLISTINT) = NUMERO
80 NEXT I
90 INPUT "QUIERE BUSCAR UN NUMERO EN LA
   LISTA";RESPUESTA$
100 WHILE RESPUESTA$ = "SI"
110    INPUT DEMANDA
120    I = 1:ENCONTRE = - 1
130    WHILE I < = LONGLISTINT AND NOT(ENCONTRE)
140       IF NROLISTA(I) = DEMANDA
            THEN POSICION = I:ENCONTRE = 0
            ELSE I = I + 1
150    WEND
160       IF NOT(ENCONTRE)
            THEN PRINT DEMANDA;"NO ESTA EN LA LISTA"
            ELSE PRINT DEMANDA;"ESTA EN LA POSICION;I
170       INPUT "QUIERE BUSCAR UN NUMERO EN LA
          LISTA";RESPUESTA$
180 WEND
190 END
```



## EJERCICIOS

Desagregar jerárquicamente hasta completar un programa en BASIC que:




1 - Dado un vector de N componentes, halle la sumatoria de sus elementos. $S = \sum_{I=1}^{N} A(I) = A(1) + A(2) + \ldots + A(N)$.

2 - Dado un vector de N componentes, halle la productoria de sus elementos, $P = \prod_{I=1}^{N} A(I) = A(1) * A(2) * \ldots * A(N)$.

3 - Dado N, genere un vector de N componentes, en el cual los elementos sean los primeros N números naturales impares.

4 - Dado N, genere un vector de N componentes, con 1 y 0 alternados.

5 - Dado un vector de N componentes, imprima la cantidad de elementos negativos del mismo.

6 - Dado un vector de N componentes, genere la sumatoria de sus elementos positivos.

7 - Dado un vector de N componentes, genere la sumatoria de los elementos de subíndice par.

8 - Dado B y un vector de N componentes, genere e imprima la productoria de los elementos impares de posición múltiplo de B.

9 - Dado un vector de N componentes, generar otro vector de N componentes, donde cada elemento sea reemplazado por el factorial del elemento original.

**10 - Dado un vector de N componentes, cuyos elementos** son 0 ó 1, genere otro vector donde el primer componente sea la sumatoria de la cantidad de 1 y la segunda la sumatorla de la cantidad de 0 del vector dado.

11 - Dado A(N) y B(N), cuyos elementos son 1 ó 0, genere dos vectores que contengan: uno, los elementos de A que corresponden 0 en B y otro los elementos de B que corresponden 1 en A.

12 - Dados dos vectores A y B de N elementos cada uno, representando dichos elementos los dígitos de dos números de N cifras (que pueden ser 0) desarrolle un algoritmo que sume y deje el resultado en el vector C(N + 1).

13 - Dado un vector de N componentes, genere e imprima otro vector de N componentes, donde cada componente sea el producto de la componente original por su subíndice.

14 - Dados A(N) y B(N), genere e imprima otro vector que contenga los elementos de las posiciones pares de A y los de las posiciones impares de B.

15 - Dado un vector de N componentes, imprima la posición del máximo elemento.

16 - Dados A(N) y B(N), cuyos elementos son 1 ó 0, genere un tercer vector que contenga 1 si ambos componentes son iguales y 0 si no lo son.

17 - Dado un vector de N componentes, imprima el producto del máximo elemento por su subíndice.

18 - Dado A(N), cuyos elementos son 1 ó 0, halle el producto de la suma de los elementos que en posición par son 1, por la suma de los elementos que en posición impar son 0.

19 - Dados A(N), genere otro vector B(2), donde la primer componente sea el elemento mayor del vector A, y la segunda el elemento que le sigue en ese orden.

20 - Dado A(N), genere otro vector B(2), donde la primer componente sea el elemento menor del vector A y la segunda componente la posición del menor elemento del vector A.

21 - Dada A(M,N), halle la sumatoria de todos sus elementos, recorriendo la matriz por filas.

22 - Dada A(M,N), halle la sumatoria de los elementos positivos.

23 - Dada A(M,N) halle la productoria de todos sus elementos, recorriendo la matriz por columnas.

24 - Dada A(M,N) halle el producto de la sumatoria de los elementos positivos, por la sumatoria de los negativos.

25 - Dada A(M,N) genere e imprima la misma matriz, donde se reemplace los elementos positivos por su factoriales.

26 - Dada A(M,N) halle la diferencia entre la sumatoria de la cantidad de elementos positivos menos la sumatoria de la cantidad de elementos negativos.

27 - Dada A(M,N), halle el producto entre la suma de elementos positivos de la diagonal principal, por la suma de los elementos negativos de la diagonal secundaria. A(1,1), A(2,2), A(3,3),...A(M,M) diagonal principal A(1,M),A(2,M-1), A(3,M-2),...A(M,1) diagonal secundaria.

28 - Dada A(M,M), halle la diferencia entre la cantidad de elementos positivos de la diagonal principal menos la cantidad de elementos negativos de la diagonal secundaria.

29 - Dada A(M,N), halle el máximo elemento.

30 - Dada A(M,N), genere un vector B(N), donde figuren los máximos elementos de cada columna.

31 - Dada A(M,N), genere un vector B(N), donde figuren las filas de los mínimos elementos de cada columna de la matriz A.

32 - Dada A(M,N), genere el vector B(M), donde figuren los mínimos elementos de cada fila.

33 - Dada A(M,N) y B(N), genere e imprima otro vector C(N), tal que la componente C(I) sea la cantidad de elementos de la columna I de la matriz A, que no son menores que B(I).

## 1. GOSUB ... RETURN

Imaginemos que tenemos un programa y que hay un conjunto de instrucciones que se repite varias veces dentro del mismo. Veamos un ejemplo:

```
10....
.........
40....
50 READ X, Y
60 Z=A*X+B*Y
70....
.........
180....
190 READ X.Y
200 Z=A*X+B*Y
210....
.........
420....
430 READ X.Y
440 Z=A*X+B*Y
450....
.........
800 END
```




Supongamos que queremos agregar líneas, además de:

```
READ X,Y
Z=A*X+B*Y
```

Lo que queremos hacer es ver los valores de X, Y, Z. Entonces tendríamos que agregar:

```
PRINT "X=";X;"Y=";Y
PRINT "Z=";Z
```

Tendríamos que modificar no sólo una sección del programa sino tres. Ingresar 3 veces cada línea sería no sólo muy aburrido sino que también deberíamos tener mucho cuidado para no equivocarnos.

Si tuviéramos en un programa un grupo de cientos de líneas que aparecieran en 20 secciones diferentes y encontraramos en ellas errores nos pasaríamos horas modificándolo.

Para no tener que escribir, en distintas secciones del programa, las sentencias que se repiten, lo ideal sería escribirlas sólo una vez y poder referirnos a ellas cada vez que lo necesitamos.

En BASIC existe una sentencia que nos permite hacer esto. Esa sentencia es GOSUB <nro de línea>.

Con la sentencia GOSUB podemos hacer que la ejecución del programa no continue en la línea siguiente sino en la línea que nosotros le indicamos.

Pero, ¿cuál es la diferencia con la sentencia GOTO? Ejecuta a partir de la sentencia indicada en el GOSUB hasta que encuentra una orden de retorno, sentencia RETURN. Por medio de ella se vuelve a la sentencia siguiente al GOSUB.

El grupo de sentencias que escribimos entre la línea indicada por la sentencia GOSUB y la sentencia RETURN es lo que llamamos una SUBRUTINA. Una subrutina es como un pequeño programita dentro de uno más grande.

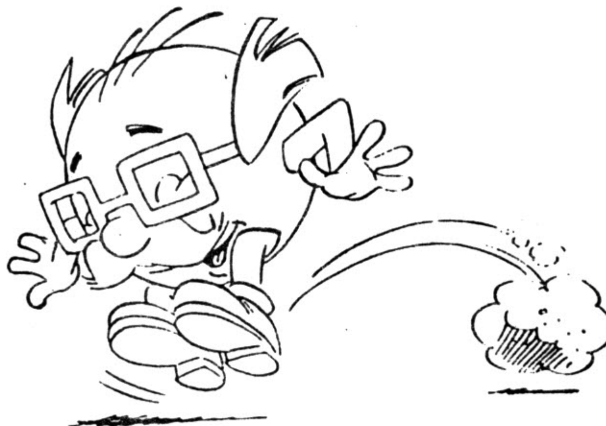
Como resolveríamos con GOSUB nuestro ejemplo anterior?

```
10 ....
..........
40 ....
50 GOSUB 810
60 ....
..........
180 ...
190 GOSUB 810
200 ...
..........
420 ...
430 GOSUB 810
440 ...
```

```
...........
800 END
810 READ X,Y
820 PRINT "X = ";X;"Y = ";Y
830 Z = A*X + B*Y
840 PRINT "Z = ";Z
850 RETURN
```

Cuál sería la secuencia de ejecución? Analicemos el programa.

* Se ejecuta de la sentencia 10 a la 40.
* Se ejecuta la sentencia 50 y salta a la 810
* Se ejecuta de la sentencia 810 a la 840
* Se ejecuta la sentencia 850 y salta a la 60
* Se ejecuta de la sentencia 60 a la 180
* Se ejecuta la sentencia 190 y salta a la 810
* Se ejecuta de la sentencia 810 a la 840
* Se ejecuta la sentencia 850 y salta a la 200
* Se ejecuta de la sentencia 200 a la 420
* Se ejecuta la sentencia 430 y salta a la 810
* Se ejecuta de la sentencia 810 a la 840
* Se ejecuta la sentencia 850 y salta a la 440
* Se ejecuta de la sentencia 440 a la 800.



¿Qué pasa? El programa se ejecuta secuencialmente hasta que encuentra la línea 50. Luego, la línea 50 indica que debe realizarse una subrutina y luego, cuando esta termine debe regresar. El programa salta a la línea 810 y ejecuta la subrutina hasta que encuentra la línea 850 y ésta ordena regresar. La ejecución contínua en la línea 60. Pasa exactamente lo mismo en las líneas 190 y 430. La única diferencia es que la ejecución regresa a otro punto cuando ejecuta la línea 850. Siempre regresa a la sentencia que le sigue a aquella que llamo a la subrutina (200 y 440). Esta es la diferencia básica con la sentencia IF ... THEN ... ELSE ... y la sentencia GOTO.

En una subrutina podemos incluir las mismas sentencias que pondríamos en cualquier otro lugar del programa.

Como todas las variables del programa están disponibles, con los mismos valores que tenían antes que la subrutina sea llamada, podemos asignarles otros valores dentro de la subrutina si eso es lo que queremos.

Dentro de un programa podemos llamar a una subrutina todas las veces que queramos y dentro de una subrutina podemos llamar a otra subrutina.

La sentencia RETURN en una subrutina hace que la ejecución del programa continúe en la sentencia que le sigue al GOSUB que la llamó. Dentro de una subrutina podemos tener más de una sentencia RETURN.

El formato de la sentencia GOSUB es:

GOSUB <nro de línea>

donde <nro de línea> es el número de línea de la primer sentencia de la subrutina.

El formato de la subrutina es:

< sentencias >

RETURN

## 2. FUNCIONES

En algún momento de nuestras vidas nos encontramos con la palabra "FUNCION". Recordemos que una función nos devuelve un valor dependiendo de otro valor dado, llamado argumento. el que le damos como entrada. Por ejemplo todos conocemos la función coseno, tangente, etc.

PECOS nos provee de funciones ya definidas y también nos permite definir nuestras propias funciones.

### A. FUNCIONES YA DEFINIDAS

Las funciones que nos provee PECOS, nos permiten evaluar muchas funciones matemáticas, rápida y fácilmente.

Podemos acceder a una función escribiendo su nombre y entre paréntesis toda aquella información que sea necesario suministrarle (ARGUMENTO).

Una vez que la función es invocada, la operación es ejecutada automáticamente sin necesidad de programación adicional.

Veamos algunas de las funciones que provee PECOS.

**ABS (X)**

Esta función devuelve el valor absoluto de X, siendo X una expresión numérica.

**Ejemplo:**




```
PRINT ABS(7*(-5))
```

Nos muestra el valor 35

## ATN(X)

Esta función devuelve el arcotangente de X en radianes, siendo X una expresión numérica.

**Ejemplo:**

```
10 INPUT X
20 PRINT ATN(X)
30 END
```

Si el valor de X que ingresamos es 3 nos muestra el valor 1.24905.

## COS(X)

Esta función devuelve el coseno de X en radianes, siendo X una expresión numérica.

Ejemplo:

```
10 X=2*COS(.4)
20 PRINT X
30 END
```

Nos muestra el valor 1.84212.

## EXP(X)

Esta función devuelve el valor de E elevado a la potencia X, siendo X una expresión numérica<= 87.3365

**Ejemplo:**

```
10 X=5
20 PRINT EXP(X-1)
30 END
```

Nos muestra el valor 54.5982.

**FIX(X)**

Esta función devuelve la parte entera de X, siendo X una expresión numérica.

**Ejemplo:**

| | |
|---|---|
| PRINT FIX(58.75) | nos muestra el valor 58 |
| PRINT FIX( – 58.75) | nos muestra el valor – 58 |

**INT(X)**

Esta función devuelve el valor del mayor entero que es < = X, siendo X una expresión numérica.

**Ejemplo:**

| | |
|---|---|
| PRINT INT(99.89) | nos muestra el valor 99 |
| PRINT INT( – 12.11) | nos muestra èl valor – 13 |

**LOG(X)**

Esta función devuelve el logaritmo natural de X, siendo X una expresión numérica mayor que cero.

**Ejemplo:**

| | |
|---|---|
| PRINT LOG(45/7) | muestra el valor 1.86075 |

SGN(X)

Esta función permite determinar el signo de X, siendo X una expresión numérica.

Si X >0, SGN(X) nos devuelve 1.
Si X = 0, SGN(X) nos devuelve 0.
Si X < 0, SGN(X) nos devuelve − 1.

SIN(X)

Esta función devuelve el seno de X en radianes siendo X una expresión numérica.

Ejemplo:

```
PRINT SIN(1.5)
```

muestra el valor 0.997495

SQR(X)

Esta función devuelve la raíz cuadrada de X, siendo X una expresión numérica.

Ejemplo:

```
10 FOR X = 10 TO 25 STEP 5
20    PRINT X,SQR(X)
30 NEXT X
40 END
```

Cuando ejecutamos este programa mostrará los siguientes valores:

| | |
|---|---|
| 10 | 3.16228 |
| 15 | 3.87298 |
| 20 | 4.47214 |
| 25 | 5 |

### TAN(X)

Esta función muestra el valor de la tangente de X en radianes, siendo X una expresión numérica.

**Ejemplo:**

```
Y = Q*TAN(X)/2
```

Antes de seguir adelante veamos que es el código de caracteres ASCII. Cuando escribimos una cadena de caracteres, los caracteres que forman la cadena no se almacenan en la memoria como caracteres sino como una secuencia codificada de números. Cada letra, dígito y caracter especial está representado por un único número. Y el conjunto de estos números forman el código ASCII.

## Código de caracteres ASCII

| Código ASCII | Caracter | Código ASCII | Caracter | Código ASCII | Caracter |
|---|---|---|---|---|---|
| 000 | NUL | 043 | + | 086 | V |
| 001 | SOH | 044 | , | 087 | W |
| 002 | STX | 045 | – | 088 | X |
| 003 | ETX | 046 | . | 089 | Y |
| 004 | EOT | 047 | / | 090 | Z |
| 005 | ENQ | 048 | 0 | 091 | [ |
| 006 | ACK | 049 | 1 | 092 | \ |
| 007 | BEL | 050 | 2 | 093 | ] |
| 008 | BS | 051 | 3 | 094 | ^ |
| 009 | HT | 052 | 4 | 095 | < |
| 010 | LF | 053 | 5 | 096 | ` |
| 011 | VT | 054 | 6 | 097 | a |
| 012 | FF | 055 | 7 | 098 | b |
| 013 | CR | 056 | 8 | 099 | c |
| 014 | SO | 057 | 9 | 100 | d |
| 015 | SI | 058 | : | 101 | e |
| 016 | DLE | 059 | ; | 102 | f |
| 017 | DC1 | 060 | < | 103 | g |
| 018 | DC2 | 061 | = | 104 | h |
| 019 | DC3 | 062 | > | 105 | i |
| 020 | DC4 | 063 | ? | 106 | j |
| 021 | NAK | 064 | @ | 107 | k |
| 022 | SYN | 065 | A | 108 | l |
| 023 | ETB | 066 | B | 109 | m |
| 024 | CAN | 067 | C | 110 | n |
| 025 | EM | 068 | D | 111 | o |
| 026 | SUB | 069 | E | 112 | p |
| 027 | ESCAPE | 070 | F | 113 | q |
| 028 | FS | 071 | G | 114 | r |
| 029 | GS | 072 | H | 115 | s |
| 030 | RS | 073 | I | 116 | t |
| 031 | US | 074 | J | 117 | u |
| 032 | SPACE | 075 | K | 118 | v |
| 033 | ! | 076 | L | 119 | w |
| 034 | " | 077 | M | 120 | x |
| 035 | # | 078 | N | 121 | y |
| 036 | $ | 079 | O | 122 | z |
| 037 | % | 080 | P | 123 | \| |
| 038 | & | 081 | Q | 124 | \| |
| 039 | ' | 082 | R | 125 | \| |
| 040 | ( | 083 | S | 126 | – |
| 041 | ) | 084 | T | 127 | DEL |
| 042 | * | 086 | U | | |

El código ASCII está en decimal.
LF = Line Feed, FF = Form Feed, CR = Carriage Return, DEL = Rubout

Ahora, que sabemos que es el código ASCII, podemos seguir adelante con otro conjunto de funciones ya definidas.

**ASC(X$)**

Esta función devuelve un valor numérico, que es el código ASCII del primer caracter de la cadena X$.

Ejemplo:

```
10X$ = "PRUEBA"
20 PRINT ASC(X$)
30 END
```

Cuando ejecutemos este programita nos mostrará el valor 80.

**CHR$(I)**

Esta función devuelve el caracter cuyo código ASCII es I, siendo I una expresión entera. Se usa habitualmente para transmitir un caracter especial a la terminal.

Ejemplo:

| | |
|---|---|
| PRINT CHR$(66) | muestra el caracter B. |
| PRINT CHR$(26) | borra lo que hay en la pantalla. |

**INSTR([I] ,X$,Y$)**

Esta función busca la primer ocurrencia de la cadena Y$ en la cadena X$ y devuelve la posición donde la encontró. Podemos indicarle desde que posición queremos que comience la búsqueda utilizando la expresión entera I, (0 <= I <= 255). Si I es mayor que la longitud de la cadena X$, o si la cadena Y$ no es encontrada INSTR nos devuelve el valor cero.

**Ejemplo:**

```
10 X$ = "ABCDEB"
20 Y$ = "B"
30 PRINT INSTR(X$,Y$);INSTR(4,X$,Y$)
40 END
```

Cuando ejecutemos este programa va a mostrar:

2 6

LEFT$(X$,I)

Esta función devuelve una cadena con los I caracteres izquierdos de la cadena X$.

**Ejemplo:**

```
10 A$ = "PECOS ES MI NOMBRE"
20 B$ = LEFT$(A$,5)
30 PRINT B$
```

Cuando lo ejecutemos nos mostrará: PECOS

**RIGHT$(X$,I)**

Esta función devuelve una cadena con los I caracteres derechos de la cadena X$.

Ejemplo:

```
10 A$ = "PECOS ES MI NOMBRE"
20 PRINT RIGHT$(A$,6)
```

Nos mostrará: NOMBRE

**MID$(X$,I,[J])**

Esta función devuelve una cadena con los J caracteres de la cadena X$, comenzando con el caracter I de la misma. I y J deben >= 0 y <= 255. Si J no se especifica o si hay menos de J caracteres a la derecha de caracter I, devuelve todos los caracteres que haya a la derecha.

Ejemplo:

```
10 A$ = "BUENAS  "
20 B$ = "TARDES NOCHES"
30 PRINT A$;MID$(B$,8,6)
```

Nos mostrará: BUENAS NOCHES

**LEN(X$)**

Esta función devuelve el número de caracteres de la cadena X$.

Ejemplo:

```
10 A$ = "BUENOS AIRES"
20 PRINT LEN(A$)
```

Nos mostrará cuando lo ejecutemos el valor 12.

**SPACE$(X)**

Esta función devuelve una cadena con X blancos

**Ejemplos:**

```
10 FOR I=1 TO 5
20   X$=SPACE$(I)
30   PRINT X$;I
40 NEXT I
```

Cuando ejecutemos nos mostrará:

```
 1
  2
   3
    4
     5
```

**SPC(I)**

Esta función imprime I blancos en la terminal, y sólo puede usarse con la sentencia PRINT.

**STR$(X)**

Esta función devuelve la cadena que representa el valor X.

**VAL(X$)**

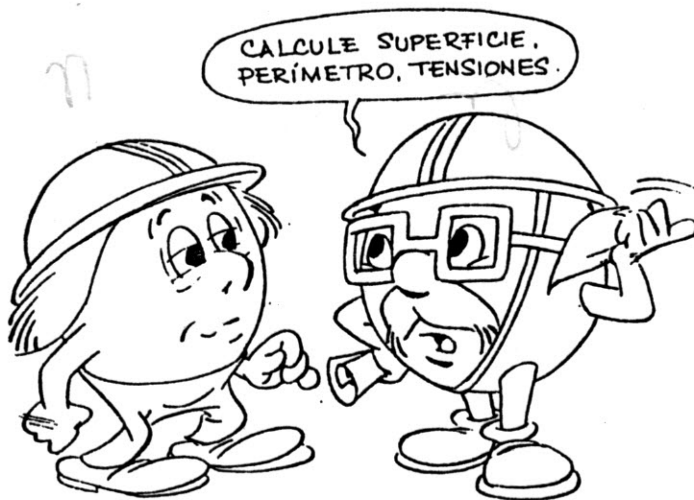
Esta función devuelve el valor numérico de la cadena X$. Si el primer carácter de la cadena X$ es +, −, & o un dígito VAL(X$)=0.

## B. FUNCIONES DEFINIDAS POR EL PROGRAMADOR

Supongamos que escribimos un programa que usa frecuentemente la fórmula del área de un triángulo (BASE*ALTURA/2). En lugar de repetir la fórmula cada vez que la necesitamos, podemos definir una función que lo haga por nosotros. Veamos un ejemplo:

```
10 DEF FNAREA(BASE,ALTURA) = BASE*ALTURA/2
20 INPUT "CUAL ES LA BASE DEL TRIANGULO",B
30 INPUT "CUAL ES LA ALTURA DEL TRIANGULO",H
40 A = FNAREA(B,H)
50 PRINT A;"ES EL AREA DE SU TRIANGULO"
60 END
```

Ahora, tenemos una función AREA, con dos argumentos, el primero es la BASE y el segundo la ALTURA del triángulo.




Podemos usar esta función en cualquier sección del programa sin volver a escribirla cada vez, sólo tenemos que invocarla por su nombre (FNAREA) y darle los valores de los argumentos.

Supongamos que conocemos la BASE y la ALTURA de dos triángulos y lo que queremos saber es que relación existe entre ambas áreas. ¿Podríamos modificar el ejemplo anterior? Probemos:

```
10 DEF FNAREA(BASE,ALTURA) = BASE*ALTURA/2
20 INPUT
"BASE Y ALTURA DEL PRIMER TRIANGULO";B1,H1
30 INPUT
"BASE Y ALTURA DEL SEGUNDO TRIANGULO";B2,H2
40 PROPOR = FNAREA(B1,H1)/FNAREA(B2,H2)
50 PRINT PROPOR; "ES LA RELACION"
60 END
```

Cuando el programa invoca a la función la primera vez el valor de B1 reemplaza a BASE en la ecuación de la función y el valor de H1 reemplaza a ALTURA. Luego, la función es evaluada, usando esos valores y nos devuelve el resultado. Lo mismo sucede la segunda vez que es invocada, y los resultados obtenidos se dividen y así obtenemos la relación entre ambas áreas.

Notemos que no hay ninguna correspondencia entre los nombres de variables que usamos en la sentencia DEF FN y los nombres que usamos para invocar la función. No sólo podemos usar variables cuando invocamos la función sino también expresiones.

El valor de la variable se le pasa a la sentencia DEF FN para realizar los cálculos. Las variables nombradas en la sentencia DEF FN son parámetros y corresponden a la ubicación de las variables en la invocación de la función. Por ejemplo, B1 y B2 son pasadas a BASE en la DEF FN, porque son las primeras en las respectivas llamadas de la función. De igual modo H1 y H2 son pasadas a ALTURA porque son las segundas.

Una sentencia DEF FN no es ejecutable en si misma, esto significa que no se ejecuta cuando es encontrada en el programa. Al ingresarla lo que estamos haciendo es crear la definición de la función después de lo cual podemos usar la función en cualquier parte del programa, tal como la hemos definido.

El formato de la sentencia DEF FN es:

DEF FN <nombre >[<lista de parámetros >] = <definición de la función >

<nombre> debe ser un nombre de variable. Este nombre precedido por FN, se convierte en el nombre de la función.

<lista de parámetros> esta compuesta por los nombres de las variables que serán reemplazados cuando la función sea invocada. Los ítems de la lista deben estar separados por comas.

<definición de la función> es una expresión que realiza la operación de la función. Los nombres de las variables que aparecen en la expresión sirven sólo para definir la función, y no afectan a las variables del programa que tengan el mismo nombre. Una variable que se usa en la definición de la función puede o no aparecer en la lista de parámetros. Si aparece será reemplazado por el valor correspondiente cuando la función sea invocada, y sino se usará el valor que tenga en ese momento la variable.

Supongamos que queremos ordenar en forma creciente una lista de valores dados, y que queremos ver la lista de valores original y la lista de valores ordenada.

Por ejemplo: dada la lista 5, 0, – 7, 6.2, 1 queremos que quede: – 7, 0, 1, 5, 6.2.

Planteemos el problema con mayor claridad:

dado un número entero N (no mayor que 50), seguido de una lista de N números, ordenarlos en orden creciente, mostrar la lista en el orden original y la lista ordenada.

Para resolver este problema, es obvio que necesitamos disponer de todos los valores de una sola vez, que debemos ingresarlos antes de ordenarlos y que debemos ordenarlos antes de mostrar la lista ordenada. Entonces los pasos a seguir son:

1. Leer la lista de valores
2. Ordenar la lista
3. Mostrar la lista ordenada

Nos queda por decidir cuando mostrar la lista de valores sin ordenar. Tenemos dos opciones: mostrarla antes de ordenarla o después. La más obvia es mostrarla antes de ordenarla. Entonces nos quedaría:

1. Leer la lista de valores
2. Mostrar la lista sin ordenar
3. Ordenar la lista
4. Mostrar la lista ordenada

Podemos combinar las dos primeras tareas en una sola, o sea:

1. Leer y mostrar la lista de valores
2. Ordenar la lista
3. Mostrar la lista ordenada

Para conservar la lista de valores necesitaremos un arreglo que llamaremos: LISTA y además una variable que nos dé la longitud del mismo: LONGLISTA.

Resolvamos el primer problema, o sea "LEER Y MOSTRAR LA LISTA DE VALORES"

Leer LONGLISTA

Si LONGLISTA < 0 o LONGLISTA > 50 entonces

Imprimir "LONGITUD INAPROPIADA"; LONGLISTA
Ir a fin del programa

sino nada más

Imprimir "LISTA EN EL ORDEN INICIAL"

Mientras I < = LONGLISTA

Leer LISTA(I)
Imprimir LISTA(I)
I = I + 1

repita

En BASIC esto será:

```
INPUT LONGLISTA
IF LONGLISTA<O OR LONGLISTA>50
    THEN PRINT "LONGITUD INAPROPIADA";LONGLISTA:
        GOTO fin del programa
PRINT "LISTA EN EL ORDEN INICIAL"
FOR I=1 TO LONGLISTA
    INPUT LISTA(I)
NEXT I
```

Ahora veamos el tercer problema: "MOSTRAR LISTA ORDENADA"

Imprimir "LISTA ORDENADA"
Mientras I<= LONGLISTA
Imprimir LISTA(I)
I = I + 1
repita

En BASIC esto es:

```
FOR I = 1 TO LONGLISTA
    PRINT LISTA(I)
  NEXT I
```

Lo que nos falta resolver es:

dado el arreglo LISTA (1:LONGLISTA) ordenarlo en orden creciente.

¿Como haríamos esto? Supongamos que tenemos N tarjetas numeradas colocadas en línea,¿cómo haríamos para ordenarla de manera que sus números estén en orden creciente? Una manera de hacerlo sería: examinar las tarjetas y encontrar aquella que tenga el mayor número. Tomarla e intercambiarla con la última de la línea. Luego, ignorando la última tarjeta (que está en la posición correcta) repetimos el proceso con las restantes N-1 tarjetas, o sea entre las N-1 tarjetas encontrar aquella que tenga mayor número e intercambiarla con la anteúltima y continuar con el proceso hasta que la primera y la segunda tarjeta sean intercambiadas y nos quede la lista ordenada. Transformemos esto en un algoritmo:

```
Mientras LONG < = 2

    Encontrar el máximo valor de LISTA desde
    1 hasta LONG

    Intercambiar el máximo valor con LISTA(LONG)

    LONG = LONG – 1

repita
```

Hagamos una subrutina que "ENCUENTRE EL MAXIMO" y otra subrutina que "INTERCAMBIE LOS VALORES".

En la subrutina que encuentra el máximo valor necesitaremos una variable para guardar el mayor valor: MAYORNRO y otra para guardar la posición en que este se encuentra: POSCMAYOR.

```
MAYORNRO = LISTA(1)

POSCMAYOR = 1

Para I = 2 hasta LONGLISTA
```

Si LISTA(I) > MAYORNRO entonces
MAYORNRO = LISTA(I)
POSCMAYOR = I
sino nada más
Siguiente I

La subrutina que intercambia los valores será:

```
TEMP = LISTA(LONG)
LISTA(LONG) = LISTA(POSCMAYOR)
LISTA(POSCMAYOR) = TEMP
```

El programa completo en BASIC será:

```
  10 DIM LISTA(50)
  20 INPUT LONGLISTA
  30 IF LONGLISTA <0 OR LONGLISTA > 50 THEN
       PRINT "LONGITUD INAPROPIADA:";LONGLISTA:
       GOTO 160
  40 PRINT "LISTA EN EL ORDEN ORIGINAL"
  50 FOR I = 1 TO LONGLISTA
  60     INPUT LISTA(I)
  70 NEXT I
  80 FOR LONG = LONGLISTA TO 2 STEP -1
  90     GOSUB 1000 ' encuentra el mayor valor
 100     GOSUB 2000 ' intercambia valores
 110 NEXT LONG
 120 PRINT "LISTA ORDENADA"
 130 FOR I = 1 TO LONGLISTA
 140     PRINT LISTA(I)
 150 NEXT I
 160 END
1000 '
1010 MAYORNRO = LISTA(I)
1020 POSCMAYOR = 1
1030 FOR I = 2 TO LONG
```

```
1040    IF LISTA(I)>MAYORNRO THEN
                MAYORNRO = LISTA(I):
                POSCMAYOR = I
1050 NEXT I
1060 RETURN
2000 '
2010 TEMP = LISTA(LONG)
2020 LISTA(LONG) = LISTA(POSCMAYOR)
2030 LISTA(POSCMAYOR) = TEMP
2040 RETURN
```

## EJERCICIOS:

Desagregar jerárquicamente hasta completar un programa en BASIC que:

1 - Opcionalmente:

a) liste los primeros N números naturales y determine si cada uno es o no primo.

b) determine si un número ingresado es o no primo.

(Usar una subrutina que determine si un número es o no primo.)

2 - Genere tres matrices, las imprima y determine la posición de los elementos de cada matriz que sean:

a) múltiplos de 3 y positivos

b) múltiplos de 5 y negativos

c) el mayor elemento

d) el menor elemento

(Usar subrutinas.)

3 - Genere un vector de hasta 1000 elementos, los imprima y opcionalmente:

a) busque el mayor elemento

b) busque el menor elemento

c) busque los elementos pares

d) imprima la cantidad de elementos negativos

e) imprima la cantidad de elementos positivos

f) no haga nada y termine

(Usar subrutinas.)

4 - Genere un vector y opcionalmente:

a) ordene los elementos de mayor a menor

b) ordene los elementos de menor a mayor

c) no haga nada y termine.

(Usar subrutinas.)

5 - Copie una variable de tipo string en otras tres (separándolas por los blancos).

6 - Ponga el último caracter de una variable de tipo string al principio.

7 - Cuente cuantas veces está contenido un substring en otro.

8 - Dado el arreglo A$, cuyos elementos son: A$(1) = "I"; A$(2) = "V"; A$(3) = "X"; A$(4) = "L"; A$(5) = "C", escriba los números romanos hasta N, siendo N < = 300.

(Obviar los problemas de valores menores a izquierda, o sea, escribir "IIII" para el número 4, en vez de "IV".)

## QUE ENTENDEMOS POR PROBAR UN PROGRAMA?

Se dice que "probar un programa es demostrar que no tiene errores" o que "el propósito de la prueba es mostrar que el programa realiza correctamente las funciones esperadas" o que "probar es el proceso que lleva a confiar que un programa hará lo que se supone debe hacer". Sin embargo, la causa principal de que las pruebas resulten deficientes radica en el hecho de considerar una definición incorrecta de la palabra "probar", porque describe casi lo opuesto de lo que debemos considerar como prueba de un programa.

Dejando de lado por un momento las definiciones, pensemos que probar significa aumentar su calidad o su confiabilidad, lo que a su vez, implica encontrar y eliminar errores. De esto se desprende que no debemos probar un programa para mostrar que funciona, es más conveniente partir de la suposición que el programa contiene errores y probarlo para encontrar tantos como sea posible.

Una buena definición es: "LA PRUEBA ES EL PROCESO DE EJECUTAR UN PROGRAMA CON EL FIN DE ENCONTRAR ERRORES".




Ya que los seres humanos tendemos a orientarnos hacia la obtención de metas, el establecer una meta apropiada tiene gran importancia desde el punto de vista psicológico. Si nuestra meta es demostrar que el programa no tiene errores, estaremos subconcientemente orientados hacia este fin, es decir, tenderemos a seleccionar datos de prueba con baja probabilidad de hacer que el programa falle. En cambio, si nos proponemos desmostrar que el programa tiene fallas, nuestros datos de prueba tendrán mayor probabilidad de lograrlo.

Esta definición implica que la prueba es un proceso "destructivo", lo que explica porque con frecuencia resulta tan difícil. La mayor parte de nosotros posee una visión constructiva y no destructiva de la vida: tendemos a hacer cosas en lugar de destruirlas. Esta definición de prueba tiene también consecuencias respecto de como debemos diseñar los casos de prueba y acerca de quien debe y quien no debe probar un programa dado.

Otro modo de reforzar la definición adecuada de prueba es analizar los usos de las palabras "exitosos" y "no exitoso" y en particular la forma de clasificar los resultados de los casos de prueba. En general, se considera un caso de prueba que no encuentra un nuevo error como una corrida exitosa, mientras que se estima como no exitosa una prueba que descubre un nuevo error. A menudo este es un signo de que está empleando una definición incorrecta de prueba, porque la palabra "exitoso" denota un logro y la palabra "no exitoso" algo indeseable y molesto. Sin embargo, si partimos de la hipotesis que una prueba sin ningún error constituye un desperdicio de tiempo y dinero, el considerarla "exitosa" parece inapropiado.

Un caso de prueba que descubre un error, demuestra ser una valiosa inversión. De aquí, que otra manera de dar apoyo a nuestra hipótesis de prueba sea invertir el uso que corrientemente se le da a esos dos términos. Decimos entonces, que un caso de prueba "exitoso" es el que descubre un error y que un caso de prueba "no exitoso" es el que hace que el programa produzca un resultado correcto.

Consideremos la analogía con el caso de una persona que visita al médico porque siente una sensación de malestar general. Si el doctor le manda que se haga una serie de análisis y estos no ayudan a localizar el problema, no podremos decir que los análisis resultaron "exitosos"; fueron "no exitosos", ya que el paciente ha gastado una determinada cantidad de dinero, sigue enfermo y empezará a cuestionar la habilidad del médico para diagnosticar. Sin embargo, si los análisis determinan que el paciente "padece una úlcera péptica", decimos que la prueba es exitosa, ya que ahora el médico puede prescribir el tratamiento adecuado. En la analogía identificamos al paciente enfermo con el programa en prueba.

Podemos identificar una serie de principios que en su mayoría parecen ser intuitivamente obvios y sin embargo, con demasiada frecuencia, los ignoramos.

* "Una parte importante y necesaria de un caso de prueba es la definición del resultado esperado o salida del programa". Nuevamente éste es un hecho basado en la psicología humana. Si el resultado esperado de un caso de prueba no ha sido predefinido, existe la posibilidad de que un resultado posible pero erróneo se interprete como correcto por efecto del fenómeno que se describe por "el ojo ve lo que quiere ver". A pesar de la naturaleza destructiva del proceso de prueba, todavía persiste el deseo subconciente de tener un resultado correcto. Una forma de combatir esa tendencia es estimular un examen detallado del resultado de la prueba mediante una completa descripción de lo que esperamos a la salida del programa. Por eso, un caso de prueba debe constar de dos componentes: una descripción de los datos de entrada al programa y una descripción precisa de la salida esperada para esos datos de prueba.

* "Un programador debe evitar probar sus propios programas". Esto se debe, como dijimos anteriormente, a que la prueba de programas es un proceso destructivo. Es muy difícil para un programador que ha sido constructivo durante todo el tiempo de escritura de un programa cambiar de la noche a la mañana, y adoptar una actitud de cuestionamiento con respecto a su propio programa. Además, existe otro problema importante: el programa puede contener errores debido a la mala interpretación por parte del programador del enunciado del problema o de la especificación del programa. Si este es el caso, es de esperar que el programador tenga la misma falsa interpretación cuando intente probar su programa. Esto no significa que sea imposible para un programador llegar a probar sus propios programas, sino que la prueba se hace con mayor efectividad si la realiza otra persona.

* "Debemos inspeccionar con conciencia el resultado de cada prueba". Este es quizás el principio más obvio, pero es algo que muchas veces se deja de lado.

* "Los casos de prueba deben ser escritos tanto para condiciones de entrada inválidas e inesperadas como para condiciones válidas y esperadas". Cuando probamos un programa hay una tendencia natural a concentrarnos en la condiciones de entrada esperadas y válidas a costa de ignorar las condiciones inválidas e inesperadas. Los casos de prueba que representan condiciones de entrada inválidas e inesperadas parecen tener una mayor facilidad para detectar errores que los casos preparados para condiciones de entrada válidas.

* "Examinar un programa para comprobar que no hace lo que se supone que debe hacer es sólo la mitad del problema. La otra mitad consiste en ver si el programa hace lo que no se supone que debe hacer". Implica que los programas deben ser examinados con respecto a los efectos laterales indeseados. Por ejemplo: un programa para el pago de sueldos que produce los cheques correctos es un programa incorrecto si produce, además, cheques para empleados que no existen.

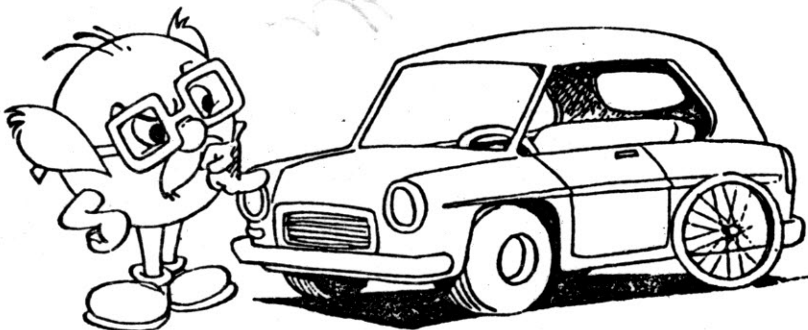
* "No debemos planificar una prueba con la suposición tácita que no encontraremos errores". Este error es cometido muy a menudo, y es una indicación del uso incorrecto de la definición de prueba, es decir, la suposición que prueba es el proceso de mostrar que un programa funciona correctamente.

## CORRECCION DE PROGRAMAS O "DEBUGGING"

El proceso de corregir un programa o "debugging" como habitualmente se lo llama, implica: localizar y corregir todos los errores que causan que un programa produzca resultados incorrectos o directamente no los produzca.

Habitualmente se piensa que nuestro trabajo está terminado cuando escribimos un programa y lo entramos en la máquina. Pero no es así, nuestro trabajo recién empieza.

El "debugging" es una de las actividades, dentro del desarrollo de programas, que los programadores menos estiman y que puede consumir mucho tiempo. Una de las razones es que la facilidad de encontrar errores en un programa está directamente relacionada con la claridad de su estructura. Los programas complicados e intrincados son mucho más difíciles de corregir.



Para ver que es lo que queremos decir, veamos cuánto tiempo nos lleva encontrar y fijar los errores en el siguiente fragmento de programa, el cual se supone que imprime el mayor de tres valores distintos: X, Y, Z.

```
 60 IF X>Y THEN 90
 70 IF Y>Z THEN 110
 80 GOTO 130
 90 IF X>Z THEN 130
100 GOTO 150
110 PRINT Y
120 GOTO 160
130 PRINT X
140 GOTO 160
150 PRINT Z
160 'continúa el programa
```

En caso que no lo encuentren, el error está en la tercer línea.

Los fragmentos de programas como éste son muy complicados de corregir, pero lamentablemente es muy común encontrar programas como este ejemplo.

Cuando estos programas son largos (cientos de líneas) resultan a menudo imposibles de seguir. Es mucho más fácil encontrar los errores cuando el programa está estructurado. Por ejemplo, consideremos el mismo fragmento (con el mismo error) codificado con IF - THEN - ELSE.

```
60 IF X>Y THEN
           IF X>Z THEN PRINT X
                  ELSE PRINT Z
           ELSE IF Y>Z THEN PRINT Y
                       ELSE PRINT X
70 ' continúa el programa
```

Así estructurado, es más fácil encontrar el error porque es más claro ver las condiciones que debemos buscar para llegar a una sección particular del programa.

Otra razón por la que los programadores encuentran dificultades durante el "debugging" es que este no se enseña con la misma sistematización que caracteriza la enseñanza de algoritmos y codificación. Esto último se enseña formalmente, con gran organización, con reglas y muchos ejemplos.

Veamos algunos de los principios que deberían seguirse para el "debugging", al igual que hicimos con las pruebas de programas, muchos de estos principios son de índole psicológica e intuitivamente obvios, a pesar de lo cual se los ignora.

Ya que el proceso de "debugging" consta de dos partes (búsqueda y corrección de los errores) veremos dos conjuntos de principios.

## PRINCIPIOS DE BUSQUEDA DE LOS ERRORES



* "Pensar". El proceso de "debugging" consiste en resolver un problema, el método más efectivo para hacerlo es analizando la información asociada con los síntomas del error. Para realizar el proceso en forma eficiente debemos ser capaces de localizar la mayoría de los errores sin necesidad de usar la máquina.

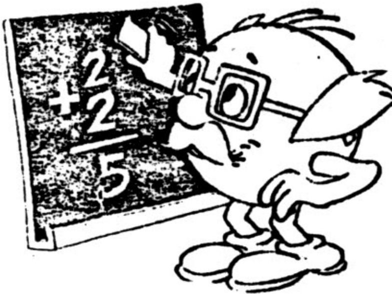
* "Si arribamos a una incomprensión, descansar". A menudo le asignamos a la inspiración el trabajo subconciente que nuestra mente efectúa mientras nos ocupamos en otra cosa como comer, caminar o ver una película. Si no podemos resolver el problema en un tiempo razonable, conviene que lo dejemos y trabajemos en otra cosa, ya que nuestra eficiencia mental de todos modos ha disminuido. Después de "olvidarlo" por un tiempo, puede ocurrir que el subconciente haya resuelto el problema o la mente consciente esté suficientemente despejada como para analizar nuevamente los síntomas.

* "Si arribamos a una incomprensión, describirle el problema a otra persona". Haciendo esto, descubriremos, probablemente algo nuevo. Es frecuente que la simple descripción del problema a otra persona haga que encontremos súbitamente la solución, sin que sea necesario su ayuda.

• "Evitemos experimentos. Usémoslos como último recurso". El error más común es tratar de resolver un problema mediante cambios exploratorios en el programa (por ejemplo: "no sé qué está mal, así que voy a cambiar la sentencia IF para ver que pasa"). Esto ni siquiera podemos considerarlo como "debugging", más bien es un acto de esperanza a ciegas. Este método, además de tener muy poca probabilidad de éxito, a menudo nos complica el problema porque le agregamos nuevos errores al programa.

## PRINCIPIOS PARA LA CORRECCION DE ERRORES

• "Donde existe una falla, es probable que exista otra". Cuando encontramos un error en una sección de un programa, la probabilidad de que haya otro error en la misma sección es generalmente mayor. O sea, los errores tienden a agruparse. Cuando arreglamos un error, conviene que examinemos sus alrededores observando otra circunstancia que nos pueda parecer sospechosa.




• "Arreglemos el error, no sólo un síntoma de él". Otro error común es arreglar los síntomas del error, o sólo una de sus manifestaciones, en vez de tratar de resolverlo.

• "La probabilidad de que la corrección esté bien no es del 100%". La codificación que agregamos a un programa para corregir un error no la podemos tomar como correcta. Si consideramos sentencia por sentencia, las correcciones que hacemos son mucho más propensas de contener errores que la codificación original del programa. Las correcciones debemos probarlas con más cuidado que el programa original.

• "La probabilidad que la corrección sea correcta disminuye con el tamaño del programa". La experiencia ha demostrado que la relación entre el número de errores debidos a correcciones mal hechas y los errores originales es mayor en programas grandes. En un programa grande, uno de cada seis nuevos errores encontrados ha sido cometido en una corrección anterior del programa.

Podemos distinguir tres tipos de errores:

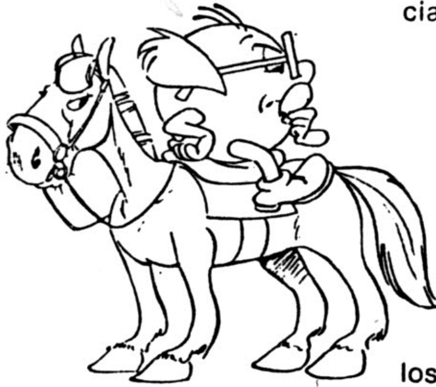
— errores de SINTAXIS
— errores de EJECUCION
— errores de LOGICA

Los errores de SINTAXIS son tal vez los más comunes en programación y los más fáciles de detectar y corregir.

Un error de SINTAXIS es simplemente una violación de las reglas gramaticales del lenguaje que estamos usando. Por ejemplo, la sintaxis de la sentencia IF en BASIC es:

IF <condición> THEN <sentencia> ! <nro de línea>
[ELSE <sentencia> ! <nro de línea> ]

Todas las sentencias IF en BASIC deben, sin excepción, tener este formato. Por lo tanto las siguientes sentencias violan las reglas de sintaxis:



```
IF A AND B PRINT A
        ELSE PRINT B
IF A AND B THEN PRINT A, PRINT B
        ELSE PRINT B
IF X + 3 ^ 2 THEN PRINT X
        ELSE PRINT Y
```

Los errores de sintaxis son más fáciles de corregir que los demás porque generalmente producen mensajes de error que nos dan una pista de que es lo que está mal en la sentencia.

Veamos un programa que contiene errores de sintaxis y tratemos de encontrarlos:

```
 10 MENOR = 0
 20 MAYOR = 100
 30 INPUT ; NUMERO
 40 SUMA = 0: NROERROR = 0
 50 FOR I = 1 NUMERO
 60      INPUT VALOR
 70      IF VALOR > = MAYOR AND VALOR < = MENOR
           THEN SUMA = SUMA + VALOR
           ELSE NROERROR = NROERROR + 1
 80 NEXT
 90 PROMEDIO = SUMA/NUMERO – NROERROR
100 PRINT "EL PROMEDIO ES:";PROMEDIO
```

Cuando el programa es gramaticalmente correcto y no produce mensajes de errores de sintaxis, no es garantía que produzca resultados correctos ya que existe la posibilidad que el programa tenga errores de ejecución.

Un error de EJECUCION es cualquiera que produzca una terminación anormal del programa.

Por ejemplo, la sentencia 90 del programa anterior:

```
PROMEDIO = SUMA/NUMERO – NROERROR
```

es sintácticamente correcta y no produce ningún mensaje de error. Sin embargo, si la variable NUMERO toma el valor 0, causa la división por cero y produce la terminación del programa y no nos daremos cuenta de esto hasta que no lo ejecutemos y termine anormalmente.

Además de la división por cero hay otros errores bastante comunes. SQR(X) o LOG(X) darán también errores de ejecución, o si los subíndices de un arreglo toman valores fuera de los límites.

Al igual que los errores de sintaxis hay errores de ejecución que producen mensajes de error.

Si corregimos los errores de sintaxis del programa anterior y lo ejecutamos ingresando como dato cero nos aparecerá el mensaje:

```
DIVISION BY ZERO
```

Por ejemplo:

```
DISCRIMINANTE = SQR(B*B – 4*A*C)
```

producirá una terminación anormal del programa si (b*b – 4*a*c) es menor que cero y la mejor manera de prevenir esto es:

```
DISCRI = B*B – 4*A*C
IF DISCRI < 0
     THEN PRINT "LAS RAICES DE LA ECUACION":
          PRINT "SON COMPLEJAS. EL PROGRAMA":
          PRINT "NO PUEDE CONTINUAR".
     ELSE DISCRIMINANTE = DISCRI
```

Para eliminar los errores de sintaxis y de ejecución generalmente invertimos una considerable cantidad de tiempo y esfuerzo. Tendremos que ejecutar el programa varias veces, simularlo manulamente, interpretar los mensajes de error y hacer las correcciones correspondientes. Todo esto nos consumirá gran cantidad de tiempo y aún así puede ser que no produzca resultados significativos. Ha llegado el momento de localizar y corregir lo que más tiempo y esfuerzo consume en el proceso de programación: los errores de LOGICA.

Un error de LOGICA es simplemente una traducción incorrecta del problema o del algoritmo. Un buen ejemplo de error de lógica es la siguiente traducción al BASIC de la fórmula para obtener las raíces de una ecuación cuadrática:

$$R = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

```
DISCRIMINANTE = (B*B) - (4*A*C)
IF DISCRIMINANTE > =0 AND A <> 0 THEN
    RAIZ1 = - B + SQR(DISCRIMINANTE) / (2*A):
    RAIZ2 = - B - SQR(DISCRIMINANTE) / (2*A)
```

Las dos sentencias de asignación contenidas en el THEN son sintácticamente aceptables y no causan ningún mensaje de error. Sin embargo, producen resultados erróneos. Estas dos líneas no son la traducción correcta de la fórmula. Lo que en realidad nos están diciendo es que:

$$R = -B \pm \frac{\sqrt{B^2 - 4AC}}{2A}$$

La manera correcta de escribir estas dos sentencias es:

```
RAIZ1 = ( - B + SQR(DISCRIMINADAMENTE))/(2*A)
RAIZ2 = ( - B - SQR(DISCRIMINADAMENTE))/(2*A)
```

Otro ejemplo de error de lógica lo encontramos en el siguiente fragmento de programa, el cual pretende encontrar la suma de varios valores. Es sintácticamente correcto, pero produce resultados erróneos.

```
10 INPUT NUMERO
20 FOR I = 1 TO NUMERO
30      SUMA = 0
40      INPUT NUMERO
50      SUMA = SUMA + NUMERO
60 NEXT I
70 PRINT "LA SUMA DE";NUMERO;"VALORES ES";SUMA
```

¿Ven el error? La inicialización de la variable SUMA no debería estar dentro del ciclo.

Ambos ejemplos nos muestran porque los errores de lógica son tan difíciles de corregir. No tenemos ninguna pista que nos indique que está pasando, lo único que sabemos es que el programa produce un valor final erróneo. Generalmente no sabemos en que sección del programa está el error y no se produce ningún mensaje que nos ayude a encontrarlo.

Si nos sentamos con lápiz y papel y simulamos manualmente el programa (como si fueramos la computadora hacemos paso por paso lo que el programa dice) puede ayudarnos.

Sin embargo, en la mayoría de los casos el programa puede ser muy complejo o el error demasiado sutil para poder localizarlo de esta manera. Lo más simple y útil es usar sentencias de salida: PRINT. A menudo tendemos a pensar que la sentencia PRINT es sólo un vehículo para mostrarnos el resultado final.

Esto es totalmente falso ya que una sentencia de salida puede usarse para mostrarnos cualquier valor, incluyendo valores intermedios, de manera que nos ayude a encontrar errores.

## EJERCICIOS

1 - Ver cuales de las siguientes sentencias contienen errores de sintaxis.

```
a - WHILE X<1 AND Y<>Z THEN PRINT A
b - FOR I TO 100
        SUMA = SUMA + 1
    NEXT I
c - IF PRUEBA AND I <= N
          THEN
          ELSE PRINT "LISTO"
d - AREA = PI*RADIO ^ 2
e - IF CUENTA = O THEN PRINT "NADA"
f - IF CUENTA < 15
          A = A + 1
          ELSE B = B + 1
```

2 - Encontrar y corregir todos los errores del siguiente programa. Decir que tipo de error es.

```
5 REM "SUMA LOS ENTEROS DESDE 1 HASTA K."
7 REM   " K ES UN DATO QUE SE INGRESA."
10 INPUT K
20 SUMA = 1
30 FOR I = 1 TO K
40       SUMA = K
50 NEXT
70 PRINT "LA SUMA DESDE 1 HASTA K ES";SUMA
```

3 - Corregir los errores de sintaxis del siguiente programa:

```
5 REM "GENERA LA SENTENCIA DE FIBONACCI:"
7 REM " X(I) = X(I = 1) + X (I – 2),X = 2,3..., X(0) = X(1) = 1."
9 REM "TERMINA CUANDO LLEGA AL LIMITE INGRESADO".
10 INPUT LIMITE
20 I = O:X = 1:PRINT I;X
30 I = 1:Y = 1 PRINT I;Y
40 WHILE Z <LIMITE
50      Z: = X + Y
60      I: = I + 1; PRINT   I ,Z
70      Y = Z
80      X = Y
90 END
100 PRINT "FIN DE LA SECUENCIA DE FIBONACCI".
110 PRINT "FUERON GENERADOS ";I;"NUMEROS".
```

4 - Después de haber hecho los cambios apropiados en el programa del ejercicio anterior, lo ejecutamos y obtenemos el siguiente resultado:

```
?100
0  1
1  1
2  2
3  4
4  8
5  16
6  32
7  64
8  128
```

a - Encontrar los errores lógicos y corregirlos.

b - Luego de corregir los errores lógicos probarlo y ver si la ejecución es correcta bajo cualquier otra circunstancia. Si no lo es, hacer los cambios necesarios que aseguren la correcta ejecución.

5 - Probar todos los programas que han codificado y corregir los errores encontrados.